三级存储综合实验报告

一、总体框架

在实际的计算机系统中不仅是 CPU 和内存模块的交互，需要主存、缓存和辅存形成的三级存储系统来与 CPU 进行数据交互，以此达到计算机的高效运行。如图 1 以 Xilinx FPGA 器件的开发板 Nexys 4 DDR Artix-7 为硬件平台，以 SD 卡作为虚拟存储器，实现了计算机的三级存储器系统。

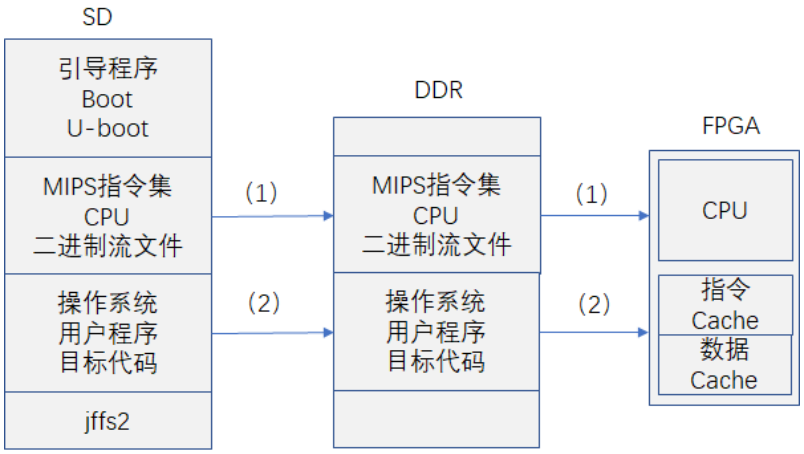


图 1 基于 Artix-7 FPGA 硬件平台的计算机存储系统

虚拟存储器存储空间的管理方式有页式和段式两种。页式管理方式把空间划分成大小相同的块；段式管理方式把空间分成可变长度的块，或称为段。前者是机械划分存储空间，后者结合程序的逻辑语义进行划分。这两种方式，对 CPU 的访存数据没有影响，不同之处在于寻址方式的差异，页式管理方式存储器的地址单一，地址字长度固定，由页号和业内位移组成；段式管理方式存储器的地址由两个字构成，一个是段号，一个是段内位移，原因是段的长度变化的。

现代计算机系统充分吸收两种方式的优点，采用段页式管理方式来管理存储空间，把一个段分成若干个页面，使虚拟存储器既具备段的逻辑单位属性，又通过以页面为单位调入主存储器，简化了虚拟地址和物理地址的转换。

虚拟存储器中的数据访存同 Cache、主存储器中的数据访存一样存在怎么映像、怎么查找、怎么替换、怎么写入的 4 个问题。计算机进行虚拟存储的访问，主要考虑低失效率这个指标，原因是对虚拟存储器访问的失效，会引起多级的连锁反应，失效开销巨大，操作系统采用全相联映像规则，允许数据块可存放在主存的任何位置。

页式管理和段式管理分别需要页表和段表的数据结构，以页号和段号作为索引，并包含待查找块的物理地址。由段内位移加上段的物理地址就构成段的最终物理地址，形成段式管理的寻址方法；而将页内位移与对应的页面物理地址拼接就构成页的最终物理地址，形成页式管理的寻址方法。

页表的索引采用的是虚拟页号，因此要求其总项数与虚拟存储空间的总页数一致。如果虚拟页面数比物理页面数多，可采用散列变换，使页表的项数减少到和主存储器的物理页面数目相等。这种方法就是反向页表法。例如，虚拟存储器地址 29 位，页大小 8KB，每个页表项为 8B，页表的大小为$\frac{2^{29}}{2^{13}} * 2^3 = 2^{19} = 512KB$，因此，一个容量为 128MB 的主存储器需要大小为 8*128MB/8KB=128KB 的反向页表来管理虚拟地址和物理地址的转换。为了减少转换时间，常采用地址转换高速缓存或称缓冲器 TLB，也称之为块表（Translation

Look-aside Buffer）。TLB 用来存放最近经常使用的页表项，可看作是页表部分内容的一个副本。在进行虚拟存储器访问时，首先会查找 TLB 表，如果 TLB 表中不命中需访问的表项，则再去访问内存中的页表。因此，TLB 表充分利用局部性原理，即，如果访问的存储器具有局部性，那么访问的存储器的地址变换同样具有局部性，也就是认为访问存储器所使用的页表项也是相对簇聚的。TLB 表中的项同 Cache 中的项相似，由标识和数据两部分组成，标识中存放的是虚拟地址的一部分，数据部分中存放物理页帧号、有效位、保护信息与其他辅助信息。在修改页表时，操作系统严格控制 TLB 表中没有该页表的表项副本，以保证 TLB 表和页表保持一致。

二、要求

构建三级存储系统，在整个执行的过程中，代码段地址和存储器地址进行统一编址，所有的代码、数据和存储器都在逻辑地址空间中有一个 32 位的地址。三级存储系统的统一编址如图 2 所示。

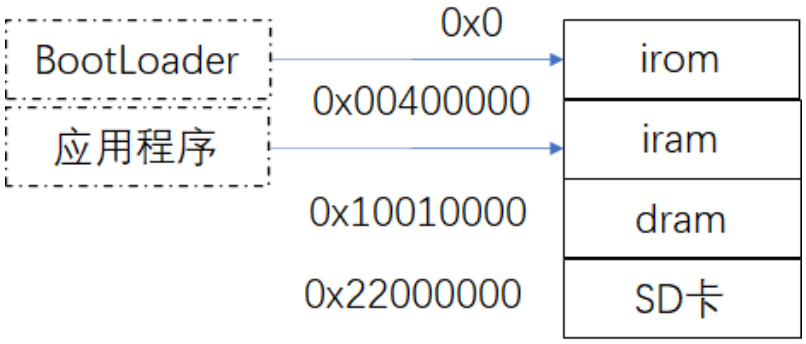


图 2 三级存储系统的统一编址

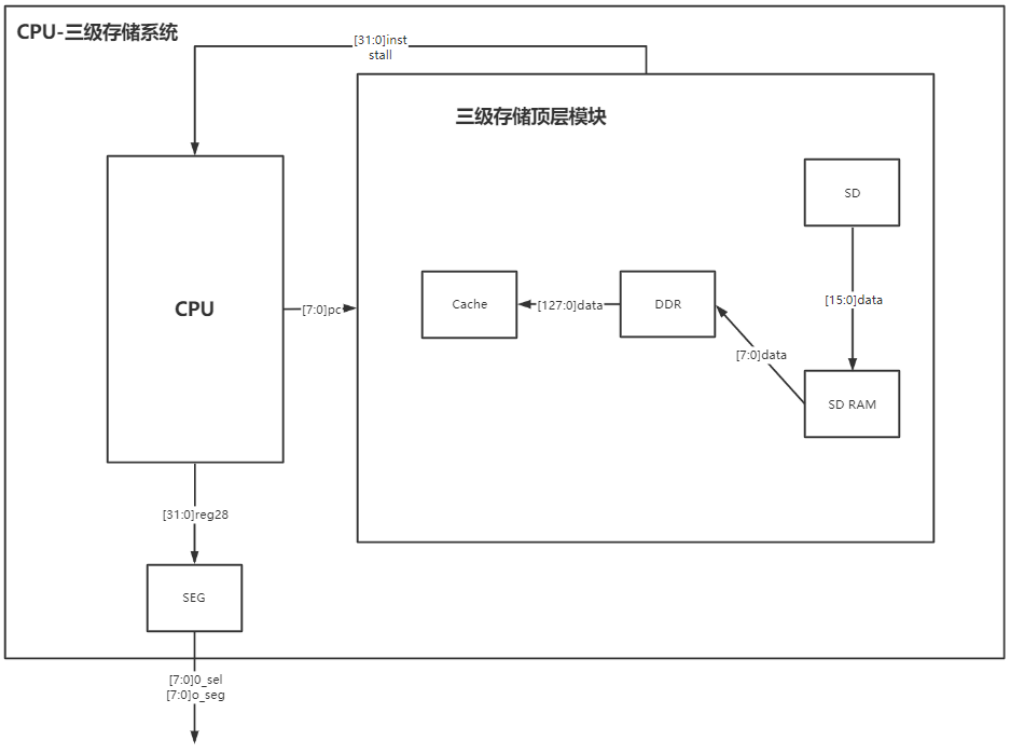
SD 卡中存放流水线 CPU 的二进制流，以及用户程序，N4 板上电自动完成如下的任务：

1） 采用跳线的方式，FPGA 自动从 SD 卡中获取流水线 CPU 的二进制流,并运行该二进制流，使 FPGA 成为 CPU。

2） CPU 再按照三级存储的方式访问 SDRAM，再由 SDRAM 从 SD 卡中把用户程序的目标代码调入到 SDRAM，再由 CPU 把 SDRAM 中的用户程序目标代码调入到片内 CACHE 加以运行。

三、实现方法

1、系统设计整体模块图



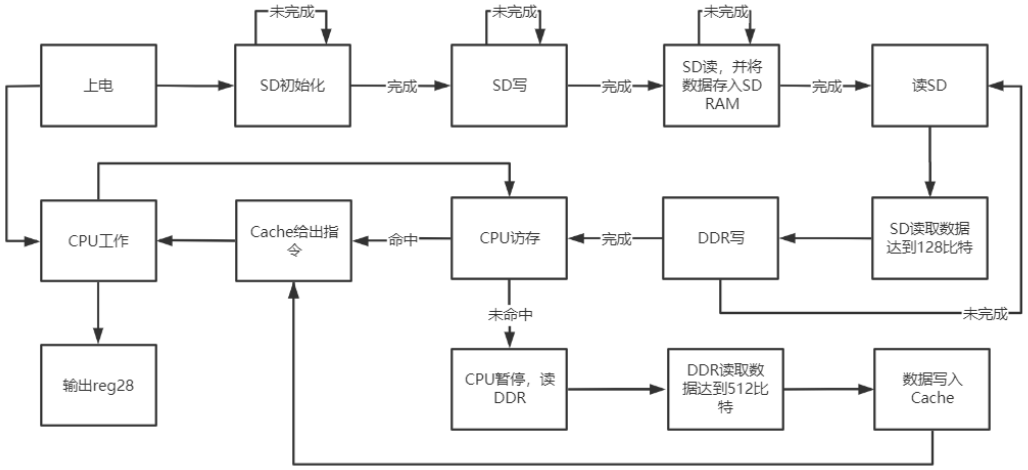
2、系统工作数据流说明

三级存储子系统与 CPU 协调工作，CPU 会告诉存储子系统访问的地址，存储子系统会通知 CPU 工作状态是否能继续和取出的数据。

开发板上电后会启动 SD 模块工作，实现 SD 卡初始化，并将相应的 MIPS 指令写入 SD 卡第 20 个扇区，然后从 SD 卡的第 20 个扇区处读出 MIPS 指令，放入 SD RAM 中，然后三级存储顶层模块会将 SD RAM 中的数据搬到 DDR 中。

CPU 初始状态会读取 0 地址，因为一开始，Cache 中是没有数据的，因此不命中，传递出相应信号，CPU 暂停流水线，等待。同时该信号通知三级存储顶层模块，将 DDR 的数据搬入 Cache。之后 CPU 开始正常工作，如果之后发生 Cache 不命中的情况，也如上所述 CPU 暂停、将 DDR 的数据搬入 Cache，再继续 CPU 工作。

整个上述过程，状态转换如下图所示：

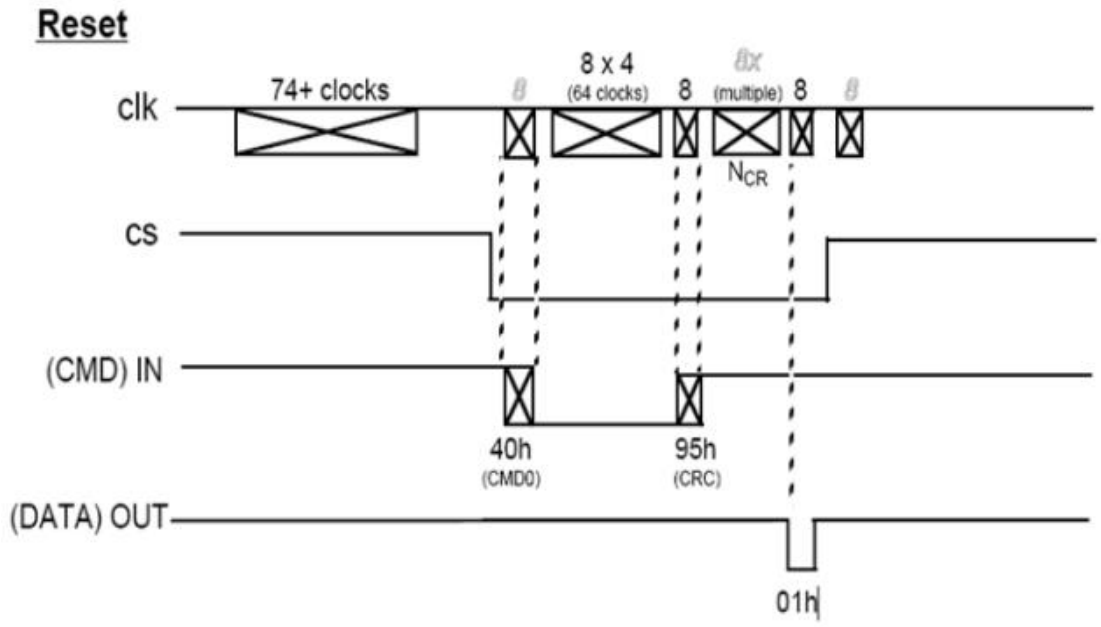


3、三级存储子系统设计

3.1SD 设计

本次实验中，SD 卡采用 SPI 协议进行通信，具体通信方式如下所述。

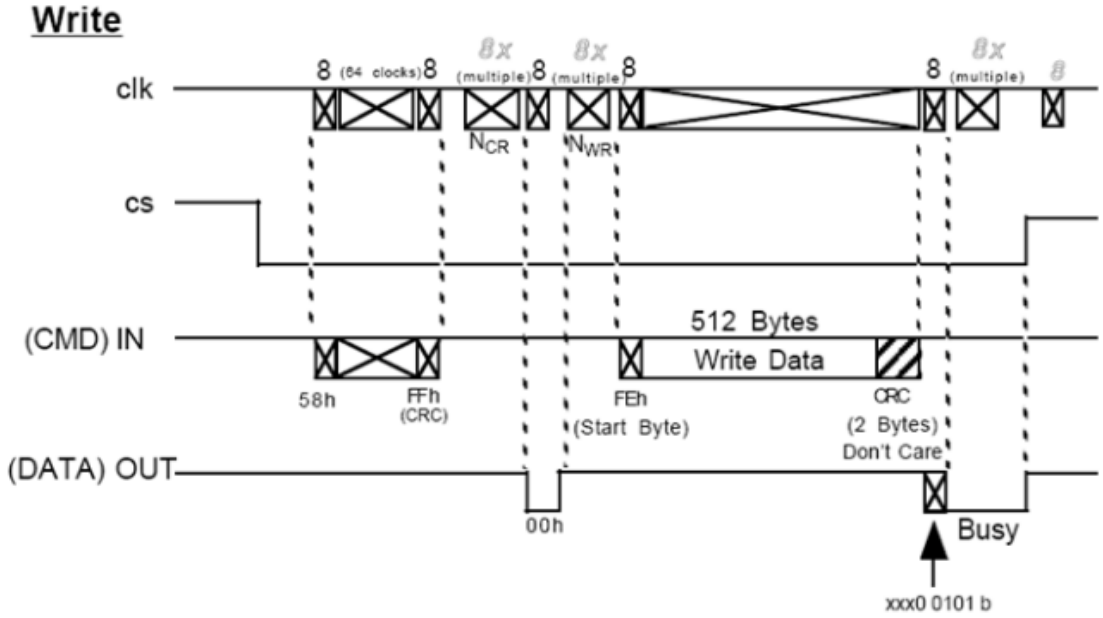
（1）SD 初始化：



由上图可知，SD 初始化模块共由以下几部分操作：

- 发送至少 74 个以上的同步时钟，等待电压稳定
- 发送命令 CMD0，等待响应数据
- 发送命令 CMD8，查询 SD 卡的版本号
- 发送命令 CMD55，告诉 SD 卡下一次发送的命令是应用相关命令
- 发送命令 ACMD41，查询 SD 卡是否初始化完成
- 初始化完成

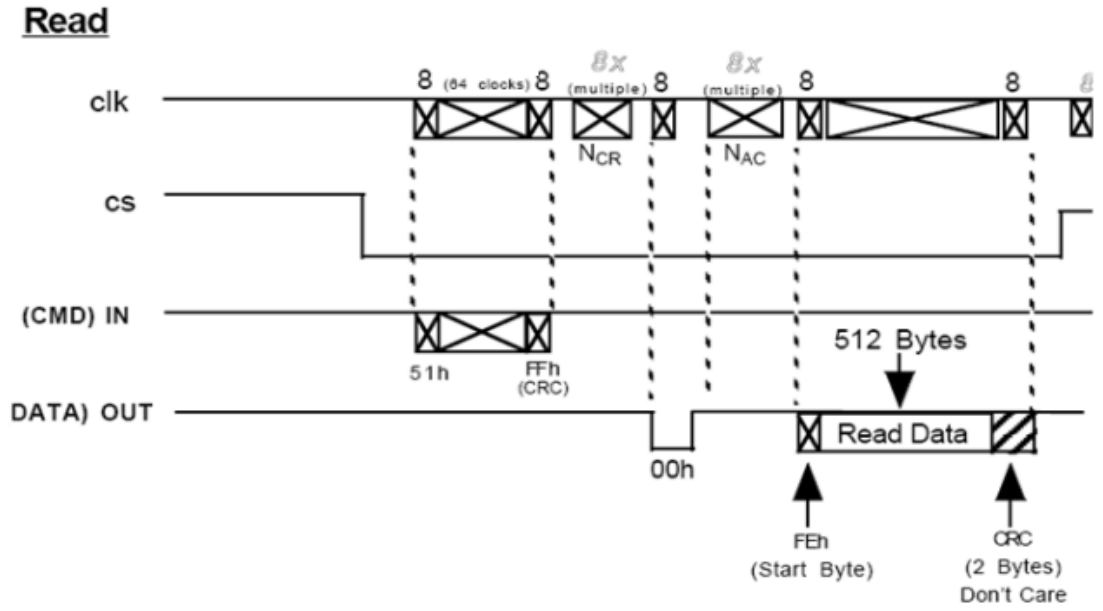
（2）SD 写：



由上图可知，SD 写模块共由以下几部分操作：

- 写空闲，等待写指令
- 发送命令 CMD24
- 发送 512 字节的数据
- 发送 CRC 校验
- 写完成

（3）SD 读：



由上图可知，SD 读模块共由以下几部分操作：

- 读空闲，等待读指令
- 发送命令 CMD17
- 读入 512 字节的数据
- 接收 CRC 校验

• 读完成

3.2DDR 设计

DDR 设计过程中需要使用 MIG 这个 IP 核，MIG 的配置过程参考了课程群中的 DDR2 读写例程。另外 FPGA 如果需要对 DDR 进行读写，则需要一个 DDR 的控制器。根据官方的文档，DDR 控制器的时序主要有三：

（1）首先是控制信号，如下图：

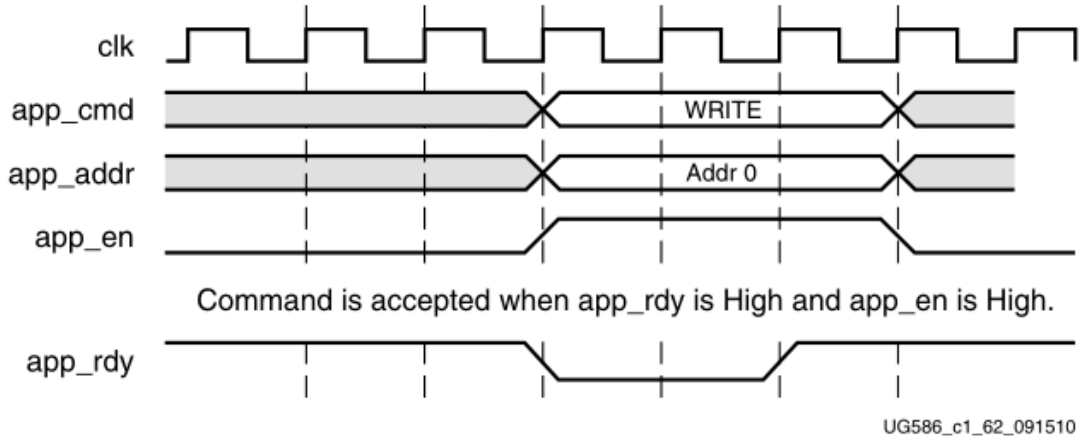


Figure 1-74: **UI Command Timing Diagram with app_rdy Asserted**

从上图可以看出，只有当 app_rdy 信号有效时，程序所发出的读写命令才会被控制器接收。这点必须注意。

（2）然后是写操作时序，如下图：

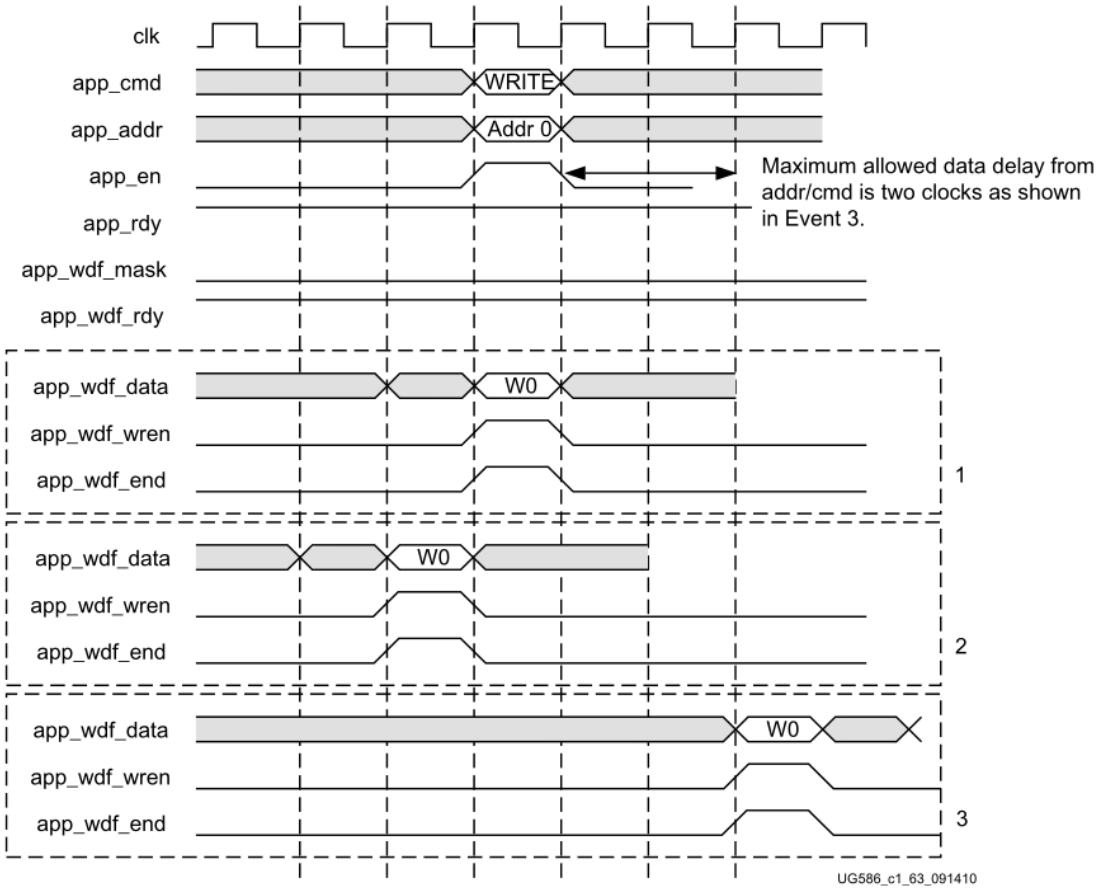


Figure 1-75: **4:1 Mode UI Interface Write Timing Diagram (Memory Burst Type = BL8)**

由图可知，在向 DDR 写数据时，需要提供写命令 app_cmd、地址 app_addr、数据 app_wdf_data 等信号，且写入的数据最多可以比 app_cmd 提前一个时钟周期有效，最迟可以比 app_cmd 晚两个时钟周期有效。

特别注意，在写数据的时候必须检测 app_rdy 和 app_wdf_rdy 信号是否同时有效，否则写入命令无法成功写入到 DDR 控制器的命令 FIFO 中，从而导致写操作失败。

（3）最后是读操作时序，如下图所示：

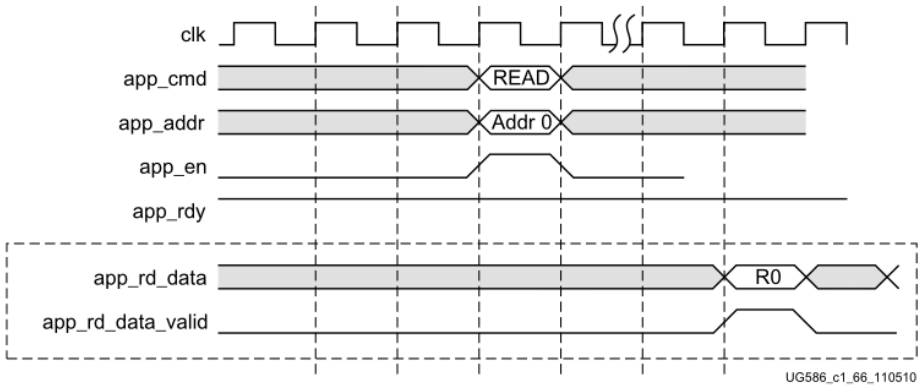


Figure 1-81: **4:1 Mode UI Interface Read Timing Diagram (Memory Burst Type = BL8)**

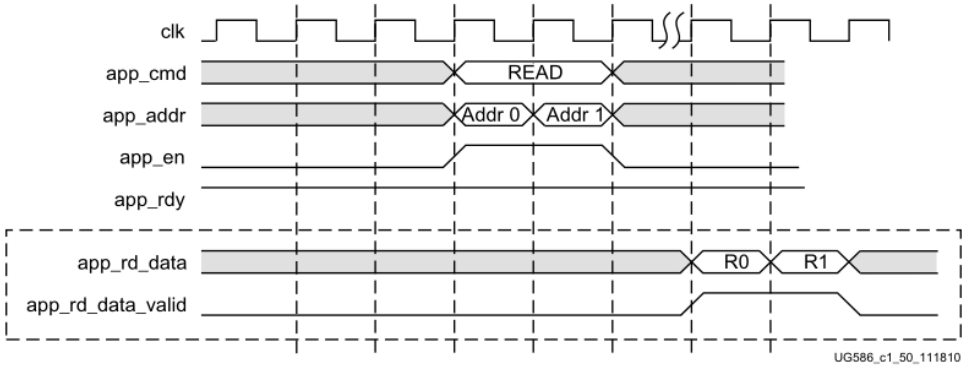


Figure 1-82: **2:1 Mode UI Interface Read Timing Diagram (Memory Burst Type = BL4 or BL8)**

读操作的时序比较简单，只需要注意 app_rdy 是否有效即可，其余不再赘述。

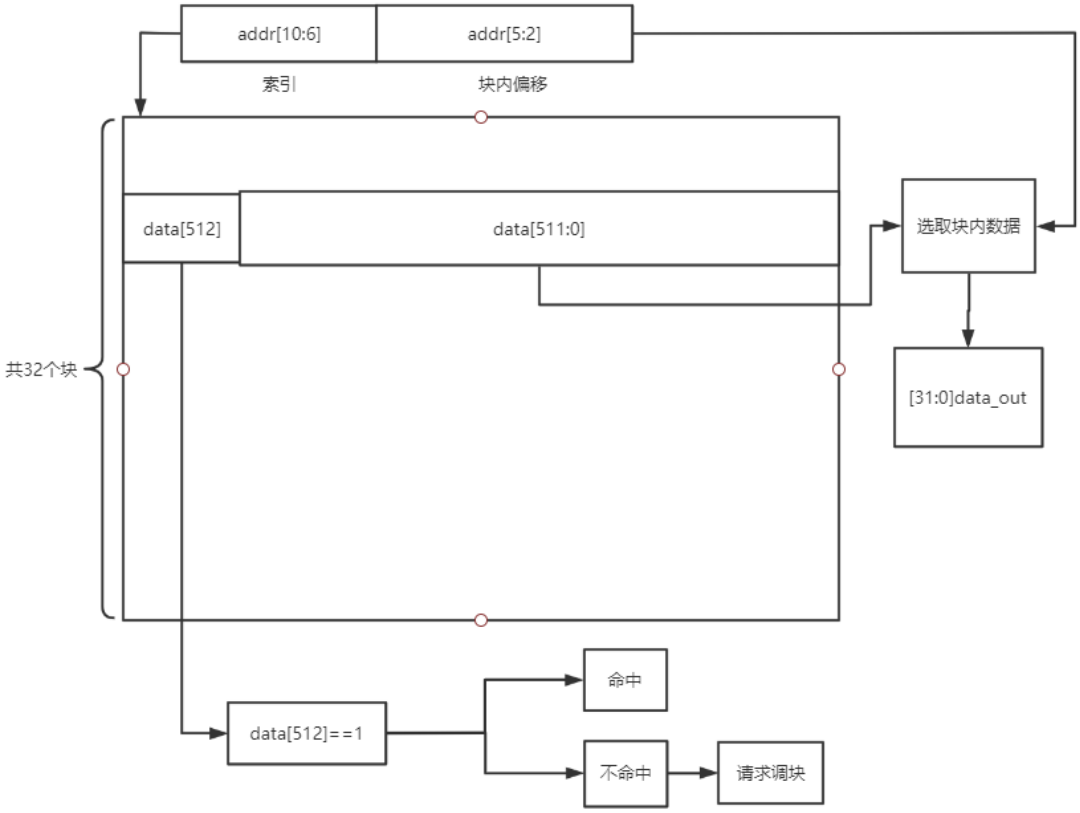
另外，存在 DDR 读取数据可能不正确的情况，猜测是由于 MIG 这个 IP 核内部设计复杂、门级较多，延迟较高，在过高的时钟频率下不能正确运行。根据实验指导书中提到的重复读取 256 次，基本可以实现正确读取。

3.3Cache 设计

在本次实验中，Cache 块的大小为 512 比特，共有 32 个 Cache 块（因为指令较少，不会完全使用这些块），采用直接映像的方式。Cache 设计为[512:0]data[0:31],每个 Cache 对应一个 RAM，这个 RAM 有 1 位有效位和 512 位数据字段，共 513 位。其中，data[512]为有效位，data[511:0]为数据字段。

当 Cache 模块传入一个地址时，应执行的操作如下：

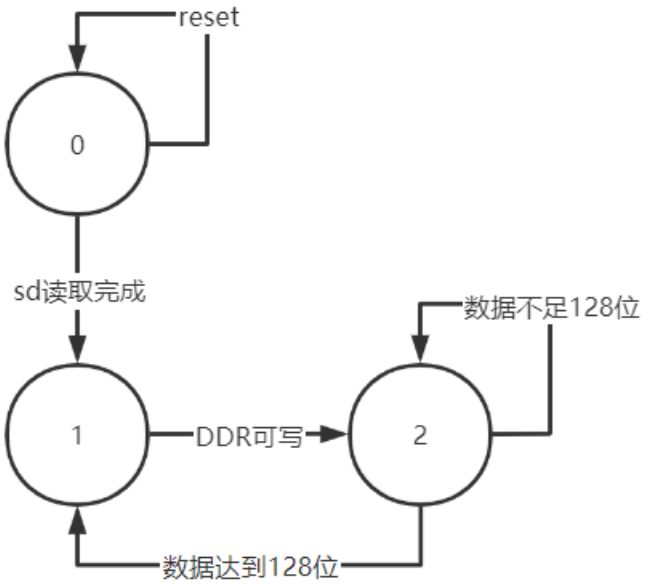
- 用访存地址的[10:6]进行 Cache 寻址,选出对应的块
- 检查 data[533]是否为 1，若为 1，则命中，否则不命中，向 DDR 请求调块
- 当 Cache 命中或收到 DDR 的数据并写入 Cache 后，根据访存地址的[5:2]选中数据并送出



3.4SD 与 DDR 间数据传递

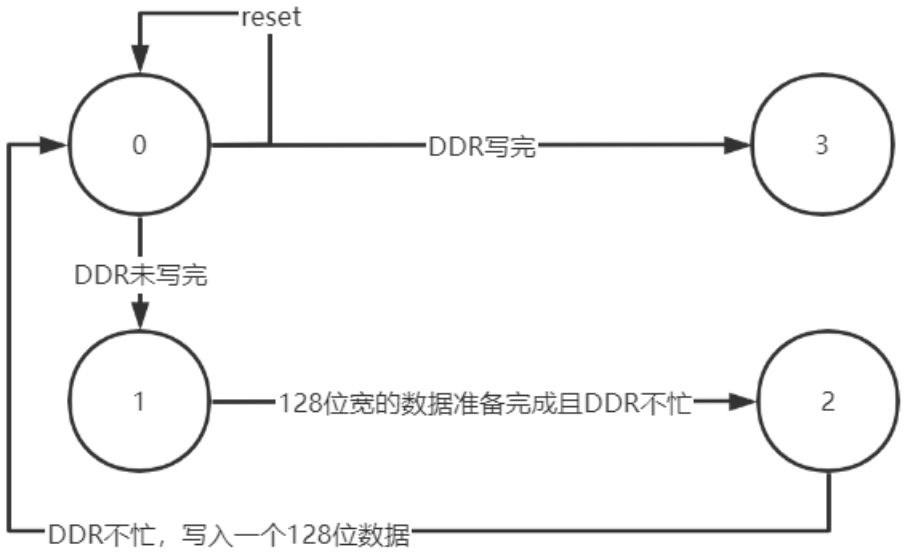
在从 SD 向 DDR 传输数据的过程中，需要设置一个 SD RAM 缓存从 SD 卡中读出的数据。因为 SD RAM 的数据宽度为 8 位，DDR 的数据宽度为 128 位，所以需要利用一个控制器来实现将 SD RAM 类型的数据转换为 DDR 类型的数据。

状态机如下：



另外，还需要一个状态机控制 DDR 的写入，将 SD 卡内存放的 512 字节的数据全部传入 DDR。

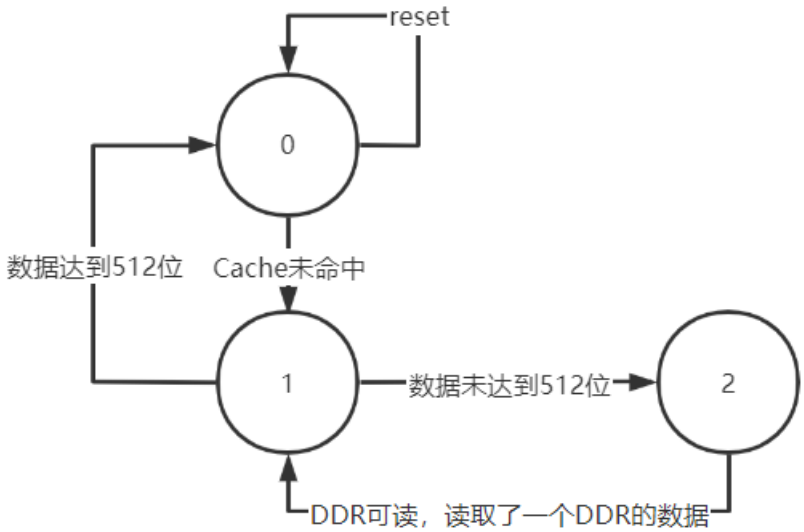
状态加如下：



3.5DDR 与 Cache 间数据传递

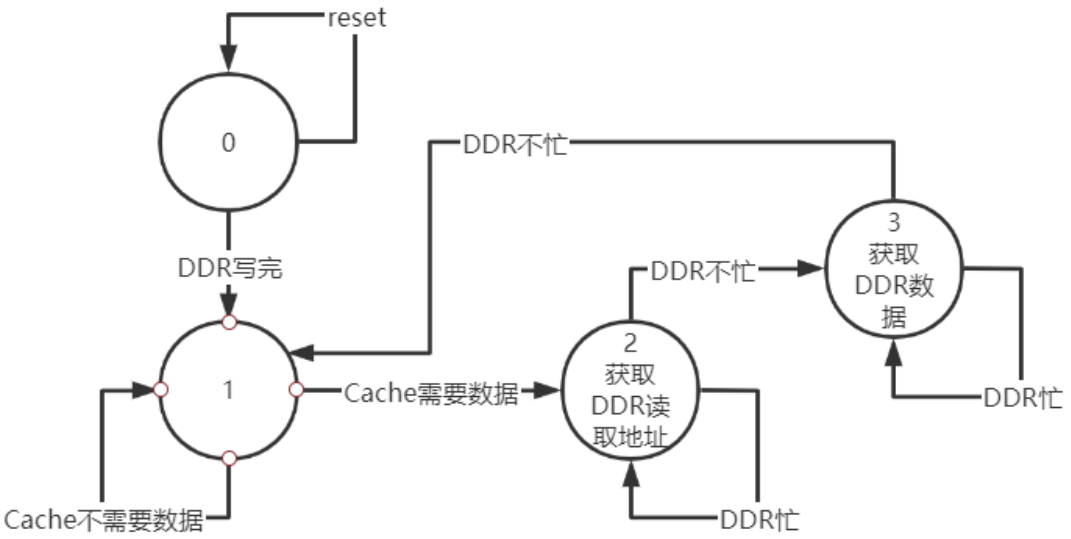
在从 DDR 向 Cache 传输数据的过程中，因为 DDR 的数据宽度为 128 位，而 Cache 的数据宽度为 512 位，所以需要利用一个控制器来实现将 DDR 类型的数据转换为 Cache 类型的数据。

状态机如下：



另外，还需要一个状态机接收 Cache 需要数据的信号，控制 DDR 的读取。

状态机如下：



4、五级流水线 CPU 改造

五级流水线 CPU 在之前动态流水线 CPU 实验中已经完成，在此不再赘述。在本次实验中，将 CPU 和三级存储系统连接起来后，需要对 CPU 进行一点小改造。因为会存在 Cache 不命中的情况，所有要从三级存储系统中传递一个 stall 信号给 CPU，当 Cache 不命中时，将 CPU 暂停，等待 Cache 写入完成并可以取出正确指令时继续工作。

四、实际运行验证

1、验证程序

在本次实验中，采用动态流水线 CPU 的验证程序，其汇编程序如下：

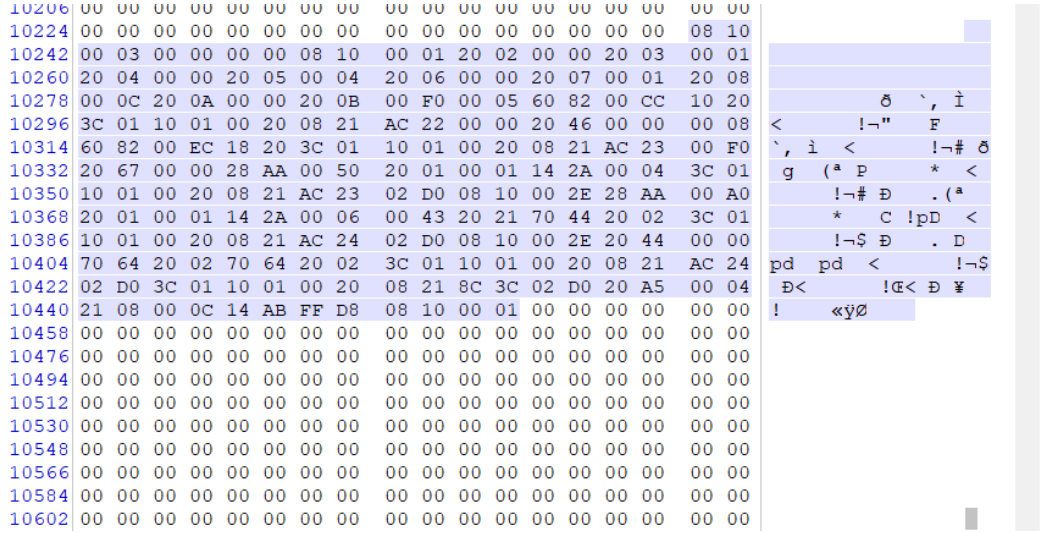
```
.data
A:.space 240
B:.space 240
C:.space 240
D:.spcce 240

.text
j main
exc:
nop
j exc

main:
addi $2,$0,0 #a[i]
addi $3,$0,1 #b[i]
```

```
addi $4,$0,0
addi $5,$0,4 #counter
addi $6,$0,0 #a[i-1]
addi $7,$0,1 #b[i-1]
addi $8,$0,0xC #counter*3
addi $10,$0,0
addi $11,$0,240

loop:
srl $12,$5,2
add $2,$6,$12 #a[i]=a[i-1]+i
sw $2, A($0)
addi $6,$2,0 #a[i-1]=a[i]

srl $12,$8,2
add $3,$7,$12 #b[i]=b[i-1]+i*3
sw $3, B($0)
addi $7,$3,0 #b[i-1]=b[i]

slti $10,$5,80 #if counter<80 $10=1
addi $1,$0,0x0001 #$1 涓哄弹绉诲姩鍏?
bne $1,$10,c1
sw $3,D($0)
j endd

c1:
slti $10,$5,160 #if counter<160 $10=1
addi $1,$0,0x0001
bne $1,$10,c2
addu $4,$2,$3
mul  $4,$2,$4
sw $4,D($0)
j endd

c2:
addi $4,$2,0
mul $4,$3,$4
mul $4,$3,$4
sw $4,D($0)

endd:
lw $28,D($0)
addi $5,$5,4
addi $8,$8,0xC
```

```
bne  $5,$11,loop

j exc
```

2、SD 写入测试

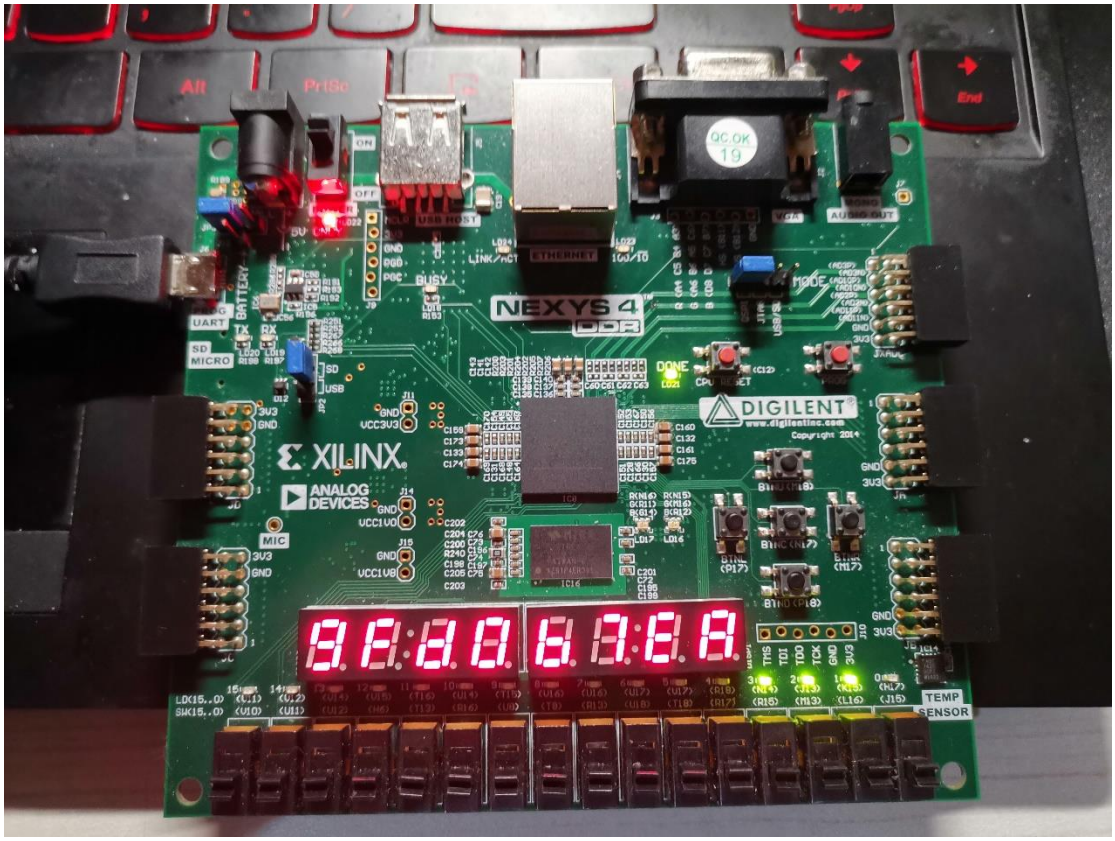
使用 Winhex 软件，观察 SD 卡的第 20 号扇区的内容，如图所示，指令的十六进制形式写入成功：



10206	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10224	00	00	00	00	00	00	00	00	00	00	00	00	00	00	08	10
10242	00	03	00	00	00	00	08	10	00	01	20	02	00	00	20	03	00	01
10260	20	04	00	00	20	05	00	04	20	06	00	00	20	07	00	01	20	08
10278	00	0C	20	0A	00	00	20	0B	00	F0	00	05	60	82	00	CC	10	20
10296	3C	01	10	01	00	20	08	21	AC	22	00	00	20	46	00	00	00	08
10314	60	82	00	EC	18	20	3C	01	10	01	00	20	08	21	AC	23	00	F0
10332	20	67	00	00	28	AA	00	50	20	01	00	01	14	2A	00	04	3C	01
10350	10	01	00	20	08	21	AC	23	02	D0	08	10	00	2E	28	AA	00	A0
10368	20	01	00	01	14	2A	00	06	00	43	20	21	70	44	20	02	3C	01
10386	10	01	00	20	08	21	AC	24	02	D0	08	10	00	2E	20	44	00	00
10404	70	64	20	02	70	64	20	02	3C	01	10	01	00	20	08	21	AC	24
10422	02	D0	3C	01	10	01	00	20	08	21	8C	3C	02	D0	20	A5	00	04
10440	21	08	00	0C	14	AB	FF	D8	08	10	00	01	00	00	00	00	00	00
10458	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10476	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10494	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10512	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10530	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10548	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10566	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10584	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
10602	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

3、下板测试

如图所示，七段数码管展示出正确的结果，左侧三个灯分别表明 SD 初始化完成、SD 写完成、SD 读完成，下板成功。



五、原程序代码与说明

1、顶层模块

作用：连接 CPU、三级存储系统和七段数码管
输入：管脚相关输入信号，主要有时钟、复位、SD 卡、DDR 相关信号等
输出：管脚相关输出信号，主要有数码管、SD 卡、DDR 相关信号等

```
`timescale 1ns / 1ps
module topcpu(
    input clk_in,
    input reset,
    output [7:0] o_seg,
    output [7:0] o_sel,
    input miso,
    output cs,
    output mosi,
    output spiclk,
    output initfinish,
    output writefinish,
    output readfinish,
/************************/
```

```
    inout [15:0]ddr2_dq,
    inout [1:0]ddr2_dqs_n,
    inout [1:0]ddr2_dqs_p,
    output [12:0]ddr2_addr,
    output [2:0]ddr2_ba,
    output ddr2_ras_n,
    output ddr2_cas_n,
    output ddr2_we_n,
    output ddr2_ck_p,
    output ddr2_ck_n,
    output ddr2_cke,
    output ddr2_cs_n,
    output [1:0]ddr2_dm,
    output ddr2_odt
/*********************/
    );
    wire cpu_stall;
    wire[31:0]reg28;
    wire[31:0]inst;
    wire[31:0]realpc;
    reg div_clk=0;
    reg[40:0]clk_cnt=0;

    always @(posedge clk_in or posedge reset)
    begin
        if(reset)
        begin
            div_clk<=0;
        end
        else if(clk_cnt==0)
        begin
            div_clk<=~div_clk;
            clk_cnt<=200000;
        end
        else
        begin
            clk_cnt<=clk_cnt-1;
        end
    end

    sccomp_dataflow cpu(
        div_clk,
        reset,
        inst,
```

```
    cpu_stall,
    realpc,//pc=realpc>>2
    reg28
);

seg7x16 seg(
    clk_in,
    reset,
    1'b1,
    reg28,
    o_seg,
    o_sel
);

wire[31:0]pc;
assign pc=realpc>>2;

wire ishit;
assign cpu_stall=~ishit;

ddrandsd threemem(
    clk_in,
    reset,
    miso,
    cs,
    mosi,
    spiclk,
    initfinish,
    writefinish,
    readfinish,

    ddr2_dq,
    ddr2_dqs_n,
    ddr2_dqs_p,
    ddr2_addr,
    ddr2_ba,
    ddr2_ras_n,
    ddr2_cas_n,
    ddr2_we_n,
    ddr2_ck_p,
    ddr2_ck_n,
    ddr2_cke,
    ddr2_cs_n,
    ddr2_dm,
```

```
        ddr2_odt,

        ishit,
        pc[7:0],
        inst
    );

endmodule
```

2、CPU 模块

作用：动态流水线 CPU
输入：存储系统给出的指令和 CPU 是否要暂停的信号
输出：PC 和 28 号寄存器存储的值

```
`timescale 1ns / 1ps
module sccomp_dataflow(
    input clk_in, //时钟
    input reset, //复位
    input [31:0]if_inst_out,
    input stall,
    output [31:0]if_pc_in,
    output [31:0]ans
    );

    reg [31:0] stage_if_inst;
    reg [31:0] stage_if_pc;
    reg [31:0] stage_if_nextpc;

    wire isbranch;
    wire [31:0] branch_pc;

    assign if_pc_in = isbranch ? branch_pc : stage_if_nextpc;

    wire [31:0] if_nextpc_out;

    wire id_stall_out;

    Instfetch instfetch(
        if_pc_in,
        if_nextpc_out
    );
    // --- CLOCK ---
    always @ (negedge clk_in)
```

```
    begin
        if(reset)
        begin
            stage_if_inst <= 0;
            stage_if_pc <= 0;
            stage_if_nextpc <= 0;
        end
        else if (!id_stall_out || !stall)
        begin
            stage_if_inst <= if_inst_out;//当前指令 inst
            stage_if_pc <= if_pc_in;//当前指令 pc，用于 ID
            stage_if_nextpc <= if_nextpc_out;//下一指令 pc，如果没有跳转，则赋给
if_pc_in
        end
    end


    reg [31:0] stage_id_alu_a;
    reg [31:0] stage_id_alu_b;
    reg [4:0] stage_id_aluchoice;
    reg [1:0] stage_id_rfaddrinchoice;
    reg [31:0] stage_id_cp0out;
    reg [31:0] stage_id_hiout;
    reg [31:0] stage_id_loout;
    reg [2:0] stage_id_rfinchoice;
    reg stage_id_rf_inallow;
    reg stage_id_hi_inchoice;
    reg stage_id_lo_inchoice;

    reg [31:0] stage_id_inst;
    reg [31:0] stage_id_rs;
    reg [31:0] stage_id_rt;

    reg [1:0] stage_id_dmem_inchoice;
    reg [2:0] stage_id_dmem_outchoice;

    //用于判断 stall
    reg [31:0] stage_exe_inst;
    reg [1:0] stage_exe_rfaddrinchoice;

    //writeback 输入
    reg [4:0]stage_id_wb_rfaddr;

    wire [31:0] id_alu_a;
```

```
wire [31:0] id_alu_b;
wire [31:0] id_rs;
wire [31:0] id_rt;
wire [4:0] id_alu_choice;
wire [1:0] id_rf_addrinchoice;
wire [31:0] id_cp0out;
wire [31:0] id_hiout;
wire [31:0] id_loout;
wire [2:0] id_rf_inchoice;
wire id_rf_inallow;
wire id_hi_inchoice;
wire id_lo_inchoice;
wire [1:0]id_dmem_inchoice;
wire [2:0]id_dmem_outchoice;

//writeback 的输出，用于 id 的输入
wire [4:0]wb_rfaddr_out;
wire [31:0]wb_rf_out;
wire wb_rf_allow_out;
wire [31:0]wb_hi_out;
wire [31:0]wb_lo_out;

//writeback 输入
wire [4:0]id_wb_rfaddr;

//把 exe_out 和 mem_out 回接
wire [31:0] exe_alu_out;
wire [31:0]mem_wb_rf;

Instdecode instdecode(
    clk_in,
    reset,
    stage_if_pc,
    stage_if_inst,
    wb_rfaddr_out, //写 regfile
    wb_rf_out,
    wb_rf_allow_out,
    wb_hi_out, //写 hi_lo
    wb_lo_out,
    stage_id_inst, //判断 stall
    stage_id_rfaddrinchoice,
    stage_exe_inst,
    stage_exe_rfaddrinchoice,
```

```
    //把 exe_out 和 mem_out 回接
    exe_alu_out,
    mem_wb_rf,

    id_alu_a,
    id_alu_b,
    id_rs,
    id_rt,
    id_alu_choice, //入 alu 和 writeback
    id_rf_addrinchoice,
    id_rf_inchoice,
    id_rf_inallow,//入 writeback
    id_hi_inchoice,
    id_lo_inchoice,//入 writeback

    isbranch,
    branch_pc,
    id_stall_out,
    id_cp0out,
    id_hiout,
    id_loout,
    id_dmem_inchoice,
    id_dmem_outchoice,
    ans,

    id_wb_rfaddr

);

// --- CLOCK ---
always @ (negedge clk_in)
begin
    if(reset)
    begin
        stage_id_alu_a <= 0;
        stage_id_alu_b <= 0;
        stage_id_aluchoice <= 0;
        stage_id_inst <= 0;
        stage_id_rs <= 0;
        stage_id_rt <= 0;
        stage_id_rfaddrinchoice <= 0;
        stage_id_cp0out <= 0;
        stage_id_hiout <= 0;
        stage_id_loout <= 0;
```

```
            stage_id_rfinchoice <= 0;
            stage_id_rf_inallow <= 0;
            stage_id_hi_inchoice <= 0;
            stage_id_lo_inchoice <= 0;
            stage_id_dmem_inchoice <= 0;
            stage_id_dmem_outchoice <= 0;

            //writeback用
            stage_id_wb_rfaddr <= 0;
        end
        else
        begin
            if (!id_stall_out || !stall)
            begin
                stage_id_alu_a <= id_alu_a;
                stage_id_alu_b <= id_alu_b;
                stage_id_aluchoice <= id_alu_choice;

                //writeback用
                stage_id_inst <= stage_if_inst;
                stage_id_rs <= id_rs;
                stage_id_rt <= id_rt;
                stage_id_rfaddrinchoice <= id_rf_addrinchoice;
                stage_id_cp0out <= id_cp0out;
                stage_id_hiout <= id_hiout;
                stage_id_loout <= id_loout;
                stage_id_rfinchoice <= id_rf_inchoice;
                stage_id_rf_inallow <= id_rf_inallow;
                stage_id_hi_inchoice <= id_hi_inchoice;
                stage_id_lo_inchoice <= id_lo_inchoice;
                stage_id_dmem_inchoice <= id_dmem_inchoice;
                stage_id_dmem_outchoice <= id_dmem_inchoice;

                //writeback用
                stage_id_wb_rfaddr <= id_wb_rfaddr;
            end
        end
    end

    reg [31:0] stage_exe_alu_out;

    wire [31:0] exe_alu_loout;

    reg [31:0]stage_exe_rt;
```

```
reg [1:0]stage_exe_dmem_inchoice;
reg [2:0]stage_exe_dmem_outchoice;

reg [2:0] stage_exe_rfinchoice;
reg [31:0] stage_exe_cp0out;
reg [31:0] stage_exe_hiout;
reg [31:0] stage_exe_loout;
reg stage_exe_rf_inallow;

//writeback 输入
reg [4:0]stage_exe_wb_rfaddr;
reg [31:0]stage_exe_wb_hiin;
reg [31:0]stage_exe_wb_loin;
wire [31:0]exe_wb_hiin;
wire [31:0]exe_wb_loin;

Execute execute(
    stage_id_alu_a,
    stage_id_alu_b,
    stage_id_aluchoice,

    //writeback
    stage_id_hi_inchoice,
    stage_id_lo_inchoice,
    stage_id_rs,

    exe_alu_out,

    //wb
    exe_wb_hiin,
    exe_wb_loin
);

// --- CLOCK ---
always @ (negedge clk_in)
begin
    if(reset)
    begin
        stage_exe_alu_out <= 0;
        stage_exe_inst <= 0;
        stage_exe_rfaddrinchoice <= 0;
        stage_exe_cp0out <= 0;
        stage_exe_hiout <= 0;
```

```
            stage_exe_loout <= 0;
            stage_exe_rfinchoice <= 0;
            stage_exe_rf_inallow <= 0;

            stage_exe_rt <= 0;
            stage_exe_dmem_inchoice <= 0;
            stage_exe_dmem_outchoice <= 0;

            //writaback用
            stage_exe_wb_rfaddr <= 0;
            stage_exe_wb_hiin <= 0;
            stage_exe_wb_loin <= 0;
        end
        else
        begin
            if(!stall)
            begin
                stage_exe_alu_out <= exe_alu_out;

                stage_exe_inst <= stage_id_inst;
                stage_exe_rfaddrinchoice <= stage_id_rfaddrinchoice;
                stage_exe_cp0out <= stage_id_cp0out;
                stage_exe_hiout <= stage_id_hiout;
                stage_exe_loout <= stage_id_loout;
                stage_exe_rfinchoice <= stage_id_rfinchoice;
                stage_exe_rf_inallow <= stage_id_rf_inallow;

                stage_exe_rt <= stage_id_rt;
                stage_exe_dmem_inchoice <= stage_id_dmem_inchoice;
                stage_exe_dmem_outchoice <= stage_id_dmem_outchoice;

                //writaback用
                stage_exe_wb_rfaddr <= stage_id_wb_rfaddr;
                stage_exe_wb_hiin <= exe_wb_hiin;
                stage_exe_wb_loin <= exe_wb_loin;
            end
        end
    end

    //writaback用
    reg [4:0]stage_mem_wb_rfaddr;
    reg [31:0]stage_mem_wb_rfin;
    reg stage_mem_rf_inallow;
    reg [31:0]stage_mem_wb_hiin;
```

```
reg [31:0]stage_mem_wb_loin;


Memory memory(
    clk_in,
    stage_exe_dmem_inchoice,
    stage_exe_alu_out,//也是writeback用
    stage_exe_rt,
    stage_exe_dmem_outchoice,

    stage_exe_rfinchoice,//writeback用
    stage_exe_wb_loin,
    stage_exe_cp0out,
    stage_exe_hiout,
    stage_exe_loout,

    mem_wb_rf
);

// --- CLOCK ---
always @ (negedge clk_in)
begin
    if(reset)
    begin
        stage_mem_wb_rfaddr <= 0;
        stage_mem_wb_rfin <= 0;
        stage_mem_rf_inallow <= 0;
        stage_mem_wb_hiin <= 0;
        stage_mem_wb_loin <= 0;
    end
    else
    begin
        if(!stall)
        begin
            stage_mem_wb_rfaddr <= stage_exe_wb_rfaddr;
            stage_mem_wb_rfin <= mem_wb_rf;
            stage_mem_rf_inallow <= stage_exe_rf_inallow;
            stage_mem_wb_hiin <= stage_exe_wb_hiin;
            stage_mem_wb_loin <= stage_exe_wb_loin;
        end
    end
end

Writeback writeback(
```

```
        stage_mem_wb_rfaddr,
        stage_mem_wb_rfin,
        stage_mem_rf_inallow,
        stage_mem_wb_hiin,
        stage_mem_wb_loin,

        wb_rfaddr_out,
        wb_rf_out,
        wb_rf_allow_out,
        wb_hi_out,
        wb_lo_out
    );

endmodule
```

3、三级存储系统

作用：作为连接 SD、DDR 和 Cache 的模块，配合 CPU 工作
输入：CPU 访存地址和管脚相关输入信号
输出：指令、CPU 是否要暂停的信号、管脚相关输出信号

```
`timescale 1ns / 1ps
module ddrandsd(
    input clk,
    input reset,
    input sd_miso,
    output sd_cs,
    output sd_mosi,
    output sd_clk,
    output sd_init_done,
    output sd_write_done,
    output sd_read_done,
/************************/
    inout [15:0]ddr2_dq,
    inout [1:0]ddr2_dqs_n,
    inout [1:0]ddr2_dqs_p,
    output [12:0]ddr2_addr,
    output [2:0]ddr2_ba,
    output ddr2_ras_n,
    output ddr2_cas_n,
    output ddr2_we_n,
    output ddr2_ck_p,
    output ddr2_ck_n,
    output ddr2_cke,
    output ddr2_cs_n,
```

```
    output [1:0]ddr2_dm,
    output ddr2_odt,
/************tempLook********/
    output ishit,
    input[7:0]pc_in,
    output[31:0]inst
    );

    wire[31:0]pc;
    assign pc={22'b0,pc_in,2'b0};

    reg write_cache;
    reg[511:0]cache_data_in;
    reg cache_need_data;
    reg[31:0]ddr_out_addr;
    reg[5:0]cache_cnt;

    cache icache(
        clk,
        reset,
        write_cache,
        pc,
        cache_data_in,
        ishit,
        inst
    );

    reg can_cache_read;
    reg[4:0]ddr_cache_state;
    reg[127:0]reg_ddr_data_out;
    wire[127:0]ddr_data_out;
    reg ddr_read_ready;

    always @(posedge clk or posedge reset)
    begin
        if(reset)
        begin
            ddr_cache_state <= 0;
            cache_need_data <= 0;
            write_cache <= 0;
            cache_cnt <= 0;
            cache_data_in <= 0;
        end
        else
```

```
    begin
        if(ddr_cache_state==0&&can_cache_read)
        begin
            write_cache <= 0;
            if(~ishit)
            begin
                ddr_cache_state <= 1;
                cache_need_data <= 0;
                ddr_out_addr <= {4'b0,pc[31:6],2'b0};
                cache_cnt <= 4;
            end
        end
        if(ddr_cache_state==1)
        begin
            if(cache_cnt==0)
            begin
                ddr_cache_state <= 0;
                write_cache <= 1;
            end
            else
            begin
                cache_need_data <= 1;
                ddr_cache_state <= 2;
            end
        end
        if(ddr_cache_state==2)
        begin
            cache_need_data <= 0;
            if(ddr_read_ready)
            begin
                ddr_cache_state <= 1;
                cache_cnt <= cache_cnt-1;
                ddr_out_addr <= ddr_out_addr+1;
                cache_data_in <= {cache_data_in[383:0],reg_ddr_data_out};
            end
        end
    end
end

wire[8:0]sd_addr;
wire[7:0]sd_data_out;

topsd sdtop(
    clk,
```

```
    ~reset,
    sd_miso,
    sd_cs,
    sd_mosi,
    sd_clk,
    sd_init_done,
    sd_write_done,
    sd_read_done,
    sd_addr,
    sd_data_out
);

wire ddr_busy;
wire ddr_done;
wire ddr_start_ready;

reg[5:0]ddr_state;
reg[31:0]result;
reg ddr_read_write;
reg[31:0]ddr_addr;
reg[127:0]ddr_data_in;

reg can_write_ddr;
reg[4:0]sd_ddr_state;
reg[127:0]reg_ddr_data_in;
reg sd_data_ready;
reg[3:0]ddr_cnt;

assign sd_addr={ddr_addr[4:0],ddr_cnt};

//把 sd_output 传递到 reg
always @(posedge clk or posedge reset)
begin
    if(reset)
    begin
        reg_ddr_data_in <= 0;
        sd_data_ready <= 0;
        ddr_cnt <= 0;
        sd_ddr_state <= 0;
    end
    else
    begin
        if(sd_ddr_state==5)
        begin
```

```
            if(sd_read_done)
                sd_ddr_state <= 0;
        end
        else if(sd_ddr_state==0)
        begin
            if(can_write_ddr)
            begin
                ddr_cnt <= 0;
                sd_data_ready <= 0;
                reg_ddr_data_in <= 0;
                sd_ddr_state <= 1;
            end
        end
        else if(sd_ddr_state==1)
        begin
            if(ddr_cnt==15)
            begin
                reg_ddr_data_in <= {reg_ddr_data_in[119:0],sd_data_out};
                sd_data_ready <= 1;
                sd_ddr_state <= 0;
            end
            else
            begin
                reg_ddr_data_in <= {reg_ddr_data_in[119:0],sd_data_out};
                ddr_cnt <= ddr_cnt+1;
            end
        end
    end
end
//sd_output 到 reg 结束


reg ddr_write_done;
always @(posedge clk or posedge reset)
begin
    if(reset)
    begin
        ddr_state <= 0;
        ddr_read_write <= 1'b0;
        result <= 32'h10000000;
        ddr_addr <= 0;
        ddr_write_done <= 0;
        can_cache_read <= 0;
    end
```

```
else if(ddr_state==0&&ddr_start_ready==1&&sd_read_done)
begin
    if(ddr_write_done==0)
    begin
        ddr_state <= 11;
        ddr_read_write <= 1'b0;
        can_write_ddr <= 1'b1;
    end
    else
    begin
        ddr_state <= 1;
        ddr_read_write <= 1'b0;
        ddr_addr <= 0;
    end
end
else if(ddr_state==11)
begin
    can_write_ddr <= 1'b0;
    if(~ddr_busy&&ddr_done&&sd_data_ready)
    begin
        ddr_state <= 12;
        ddr_read_write <= 1'b1;
        ddr_addr <= ddr_addr;
        ddr_data_in <= reg_ddr_data_in;
    end
end
else if(ddr_state==12)
begin
    can_write_ddr <= 1'b0;
    if(~ddr_busy&&ddr_done)
    begin
        ddr_state <= 0;
        ddr_read_write <= 1'b0;
        if(ddr_addr==31)ddr_write_done <= 1;
            ddr_addr <= ddr_addr+1;
    end
end
else if(ddr_state==1)
begin
    can_cache_read <= 1;
    ddr_read_ready <= 0;
    reg_ddr_data_out <= 0;
    if(cache_need_data)
    begin
```

```
            ddr_state <= 2;
            ddr_read_write <= 1'b0;
        end
        else
        begin
            ddr_state <= 1;
            ddr_read_write <= 1'b0;
        end
    end
    else if(ddr_state==2)
    begin
        if(~ddr_busy&&ddr_done)
        begin
            ddr_read_write <= 1'b0;
            ddr_addr <= ddr_out_addr;
            ddr_state <= 3;
        end
        else
        begin
            ddr_state <= 2;
        end
    end
    else if(ddr_state==3)
    begin
        if(~ddr_busy&&ddr_done)
        begin
            ddr_read_write <= 1'b0;
            reg_ddr_data_out <= ddr_data_out;
            ddr_read_ready <= 1'b1;
            ddr_state <= 1;
        end
        else
        begin
            ddr_state <= 3;
        end
    end
end

sealedDDR top_ddr(
    clk,
    reset,
    ddr_addr,
    ddr_data_in,
    ddr_read_write,
```

```
        ddr_data_out,
        ddr_busy,
        ddr_done,
        ddr_start_ready,
        //ddr signal
        ddr2_dq,
        ddr2_dqs_n,
        ddr2_dqs_p,
        ddr2_addr,
        ddr2_ba,
        ddr2_ras_n,
        ddr2_cas_n,
        ddr2_we_n,
        ddr2_ck_p,
        ddr2_ck_n,
        ddr2_cke,
        ddr2_cs_n,
        ddr2_dm,
        ddr2_odt
    );

endmodule
```

4、Cache 模块

作用：作为 Cache，与 CPU 以及 DDR 之间实现数据流动。
输入：寻址地址，是否写 Cache 的信号和写 Cache 地址
输出：Cache 是否命中的信号和给 CPU 的指令

```
`timescale 1ns / 1ps
module cache (
    input clk,
    input reset,
    input write_cache,
    input [31:0]cache_addr,
    input [511:0]cache_data_in,
    output ishit,
    output [31:0]cache_data_out
    );

    wire [512:0]block_in;
    wire [512:0]block_out;

    reg [512:0]thecache[0:31];
```

```
    assign block_in = {1'b1,cache_data_in};

    assign block_out = thecache[cache_addr[10:6]];

    assign ishit =  (block_out[512]);

    assign cache_data_out = (cache_addr[5:2]==4'b0000)?block_out[511:480]:
                            (cache_addr[5:2]==4'b0001)?block_out[479:448]:
                            (cache_addr[5:2]==4'b0010)?block_out[447:416]:
                            (cache_addr[5:2]==4'b0011)?block_out[415:384]:
                            (cache_addr[5:2]==4'b0100)?block_out[383:352]:
                            (cache_addr[5:2]==4'b0101)?block_out[351:320]:
                            (cache_addr[5:2]==4'b0110)?block_out[319:288]:
                            (cache_addr[5:2]==4'b0111)?block_out[287:256]:
                            (cache_addr[5:2]==4'b1000)?block_out[255:224]:
                            (cache_addr[5:2]==4'b1001)?block_out[223:192]:
                            (cache_addr[5:2]==4'b1010)?block_out[191:160]:
                            (cache_addr[5:2]==4'b1011)?block_out[159:128]:
                            (cache_addr[5:2]==4'b1100)?block_out[127:96]:
                            (cache_addr[5:2]==4'b1101)?block_out[95:64]:

(cache_addr[5:2]==4'b1110)?block_out[63:32]:block_out[31:0];

    always @(posedge clk or posedge reset)
    begin
        if(reset)
        begin
            thecache[0]<=0;
            thecache[1]<=0;
            thecache[2]<=0;
            thecache[3]<=0;
            thecache[4]<=0;
            thecache[5]<=0;
            thecache[6]<=0;
            thecache[7]<=0;
            thecache[8]<=0;
            thecache[9]<=0;
            thecache[10]<=0;
            thecache[11]<=0;
            thecache[12]<=0;
            thecache[13]<=0;
            thecache[14]<=0;
            thecache[15]<=0;
            thecache[16]<=0;
```

```
            thecache[17]<=0;
            thecache[18]<=0;
            thecache[19]<=0;
            thecache[20]<=0;
            thecache[21]<=0;
            thecache[22]<=0;
            thecache[23]<=0;
            thecache[24]<=0;
            thecache[25]<=0;
            thecache[26]<=0;
            thecache[27]<=0;
            thecache[28]<=0;
            thecache[29]<=0;
            thecache[30]<=0;
            thecache[31]<=0;
        end
        else
        begin
            if(write_cache)
                thecache[cache_addr[10:6]]<=block_in;
        end
    end

endmodule
```

5、DDR 模块

作用：将 SD 卡的数据存入 DDR，与 Cache 和 SD 模块之间实现数据流动。
输入：DDR 写入地址、DDR 写入数据、DDR 读还是写的信号、管脚相关输入信号
输出：DDR 输出数据、DDR 是否忙信号、DDR 是否工作完成信号、DDR 是否准备完毕信号、管脚相关输出信号

5.1DDR 顶层模块

```
`define READ 1'b0
`define WRITE 1'b1
`define NoBusy 1'b0
`define Busy 1'b1
`define DONE 1'b1
`define UNDONE 1'b0
`timescale 1ns / 1ps
module sealedDDR (
    input clk100mhz,    // Clock
```

```
    input rst,
    input [31:0]addr_to_DDR,
    input [127:0]data_to_DDR,
    input read_write,
    output  [127:0]data_from_DDR,
    output reg busy,
    output reg done,
    output  ddr_start_ready,
    /***********************/
    inout [15:0]ddr2_dq,
    inout [1:0]ddr2_dqs_n,
    inout [1:0]ddr2_dqs_p,
    output [12:0]ddr2_addr,
    output [2:0]ddr2_ba,
    output ddr2_ras_n,
    output ddr2_cas_n,
    output ddr2_we_n,
    output ddr2_ck_p,
    output ddr2_ck_n,
    output ddr2_cke,
    output ddr2_cs_n,
    output [1:0]ddr2_dm,
    output ddr2_odt
/*********************/

);

    wire [31:0] reg_addr_in;
    wire [127:0] reg_data_to_DDR;
    wire reg_read_write;
    wire clk200mhz;
    wire [1:0] rdqs_n;
    wire ack;
    wire [127:0] app_rd_data;
    wire app_rd_data_valid;
    wire app_rdy;
    reg ddr_is_ready;
    assign ddr_start_ready=ddr_is_ready;
    clk_wiz_0 clk_divider(.clk_in1(clk100mhz),.clk_out1(clk200mhz));

    ddr2_wr ddr2_wr_ins(
        .clk_in(clk100mhz),
        .rst(rst),
        //ddr2 parameter
```

```
    .ddr2_ck_p(ddr2_ck_p),
    .ddr2_ck_n(ddr2_ck_n),
    .ddr2_cke(ddr2_cke),
    .ddr2_cs_n(ddr2_cs_n),
    .ddr2_ras_n(ddr2_ras_n),
    .ddr2_cas_n(ddr2_cas_n),
    .ddr2_we_n(ddr2_we_n),
    .ddr2_dm(ddr2_dm),
    .ddr2_ba(ddr2_ba),
    .ddr2_addr(ddr2_addr),
    .ddr2_dq(ddr2_dq),
    .ddr2_dqs_p(ddr2_dqs_p),
    .ddr2_dqs_n(ddr2_dqs_n),
    .rdqs_n(rdqs_n),
    .ddr2_odt(ddr2_odt),
    .clk_ref_i(clk200mhz),
    .addr_i_32(reg_addr_in),
    .data_i(reg_data_to_DDR),
    .stb_i(reg_read_write),
    .ack_o(ack),
    .app_rd_data(app_rd_data),
    .app_rd_data_valid(app_rd_data_valid),
    .app_rdy(app_rdy)
);
parameter readInsistLoop=256;
parameter writeInsistLoop=256;
parameter ddr_startLoop=256;
reg [63:0]write_insist_count;
reg [63:0]read_insist_count;
reg [64:0]wake_count;
reg [63:0]before_rdy_count;

always @(posedge clk100mhz or posedge rst)
begin
    if(rst)
    begin
        wake_count<=0;
        busy<=`Busy;
        done<=`UNDONE;
        read_insist_count<=0;
        write_insist_count<=0;
        before_rdy_count<=0;
        ddr_is_ready<=1'b0;
    end
```

```
else if(~ddr_is_ready)
begin
    if(before_rdy_count<ddr_startLoop&app_rdy)
    begin
        before_rdy_count<=before_rdy_count+1;
    end
    else if(app_rdy)
    begin
        ddr_is_ready<=1'b1;
        busy=`NoBusy;
    end
end
else if(busy==`NoBusy&app_rdy&ddr_is_ready|done)
begin
    //Only allow accept req when notbusy
    busy<=`Busy;
    done<=`UNDONE;
    read_insist_count<=0;
    write_insist_count<=0;
end
else if(busy==`Busy&ddr_is_ready)
begin
    if(reg_read_write==`WRITE)
    begin
        if(write_insist_count>=writeInsistLoop)
        begin
            done<=`DONE;
            busy<=`NoBusy;
        end
        else
        begin
            write_insist_count<=write_insist_count+1;
        end
    end
    else if(reg_read_write==`READ)
    begin
        if(read_insist_count>=readInsistLoop&app_rd_data_valid)
        begin
            done<=`DONE;
            busy<=`NoBusy;
        end
        else
        begin
            read_insist_count<=read_insist_count+1;
```

```
                end
            end
        end
        else
        begin
            done<=`UNDONE;
        end
    end

assign data_from_DDR=app_rd_data;
assign reg_addr_in=addr_to_DDR;
assign reg_read_write=read_write;
assign reg_data_to_DDR=data_to_DDR;

endmodule
```

5.2DDR 读写模块

```
`timescale 1ns / 1ps
module ddr2_wr(
    input clk_in,
    input rst,
    input clk_ref_i,
    output ddr2_ck_p,
    output ddr2_ck_n,
    output ddr2_cke,
    output ddr2_cs_n,
    output ddr2_ras_n,
    output ddr2_cas_n,
    output ddr2_we_n,
    output [1:0] ddr2_dm,
    output [2:0] ddr2_ba,
    output [12:0] ddr2_addr,
    inout [15:0] ddr2_dq,
    inout [1:0] ddr2_dqs_p,
    inout [1:0] ddr2_dqs_n,
    output [1:0] rdqs_n,
    output ddr2_odt,

    input [31:0] addr_i_32,
    input [127:0] data_i,
    input stb_i,
    output ack_o,
```

```
output [127:0] app_rd_data,
output app_rd_data_valid,
output app_rdy
);

wire [26:0]addr_i;

assign addr_i[26:3]=addr_i_32[23:0];
assign addr_i[2:0]=3'b0;

//---MIG IP core parameter
//---user interface signals
wire [26:0] app_addr;
wire [2:0] app_cmd;
wire [2:0] app_cmd_wr;
wire [26:0] app_addr_wr;
wire app_en_wr;
wire app_en;
wire [127:0] app_wdf_data;
wire app_wdf_end;
wire app_wdf_wren;
wire app_rd_data_end;
wire app_wdf_rdy;
wire app_sr_active;
wire app_ref_ack;
wire app_zq_ack;
wire ui_clk;
wire ui_clk_sync_rst;

wire [2:0] app_cmd_re;
wire [26:0] app_addr_pe;
wire app_en_re;

assign app_en = app_en_wr | app_en_re;

assign app_cmd = (stb_i&app_en_wr) ? app_cmd_wr :
                 (~stb_i&app_en_re) ? app_cmd_re : 0;

assign app_addr=addr_i;

wire read_enable;

ddr2_write_control ddr2_write_ctr_imp(
    .clk(ui_clk),
```

```
    .reset(ui_clk_sync_rst),
    .write_addr(addr_i),
    .write_data(data_i),
    .write_stb(stb_i),
    .write_ack(ack_o),
    .read_enable(read_enable),
    //ddr2 signals
    .app_rdy(app_rdy),
    .app_wdf_rdy(app_wdf_rdy),
    .app_en(app_en_wr),
    .app_wdf_wren(app_wdf_wren),
    .app_wdf_end(app_wdf_end),
    .app_cmd(app_cmd_wr),
    .app_addr(app_addr_wr),
    .app_wdf_data(app_wdf_data)
);

ddr2_read_control ddr2_weight_load_ins(
    .clk(ui_clk),
    .reset(ui_clk_sync_rst),
    .enable(~stb_i),
    //ddr2 signals
    .app_rdy(app_rdy),
    .app_en(app_en_re),
    .app_cmd(app_cmd_re)
);

//MIG IP core
mig_7series_0 weight_ddr2_ram (
    .ddr2_addr(ddr2_addr),// Memory interface ports
    .ddr2_ba  (ddr2_ba),
    .ddr2_cas_n(ddr2_cas_n),
    .ddr2_ck_n (ddr2_ck_n),
    .ddr2_ck_p (ddr2_ck_p),
    .ddr2_cke  (ddr2_cke),
    .ddr2_ras_n(ddr2_ras_n),
    .ddr2_we_n (ddr2_we_n),
    .ddr2_dq   (ddr2_dq),
    .ddr2_dqs_n(ddr2_dqs_n),
    .ddr2_dqs_p(ddr2_dqs_p),
    .init_calib_complete(init_calib_complete),

    .ddr2_cs_n(ddr2_cs_n),
    .ddr2_dm  (ddr2_dm),
```

```
        .ddr2_odt (ddr2_odt),

        .app_addr (app_addr),// Application interface ports
        .app_cmd  (app_cmd),
        .app_en   (app_en),
        .app_wdf_data(app_wdf_data),
        .app_wdf_end (app_wdf_end),
        .app_wdf_wren(app_wdf_wren),
        .app_rd_data (app_rd_data),
        .app_rd_data_end(app_rd_data_end),
        .app_rd_data_valid(app_rd_data_valid),
        .app_rdy      (app_rdy),
        .app_wdf_rdy  (app_wdf_rdy),
        .app_sr_req   (1'b0),
        .app_ref_req  (1'b0),
        .app_zq_req   (1'b0),
        .app_sr_active(app_sr_active),
        .app_ref_ack  (app_ref_ack),
        .app_zq_ack   (app_zq_ack),
        .ui_clk       (ui_clk),
        .ui_clk_sync_rst(ui_clk_sync_rst),

        .app_wdf_mask(16'h0000),

        .sys_clk_i   (clk_in),// System Clock Ports

        .clk_ref_i   (clk_ref_i),// Reference Clock Ports
        .sys_rst     (~rst)
    );

endmodule
```

5.3DDR 写控制模块

```
`timescale 1ns / 1ps
module ddr2_write_control(
    input clk,
    input reset,
    input [26:0] write_addr,
    input [127:0] write_data,
    input write_stb,
    output reg write_ack,
    output reg read_enable,
```

```
//ddr2 signals
input app_rdy,
input app_wdf_rdy,
output reg app_en,
output reg app_wdf_wren,
output reg app_wdf_end,
output reg [2:0] app_cmd,
output reg [26:0] app_addr,
output reg [127:0] app_wdf_data
);

parameter idle = 2'b01;
parameter write = 2'b10;

reg [1:0] state;
reg [3:0] write_count;

always @(posedge clk)
begin
    if(reset)
    begin
        write_ack <= 0;
        read_enable <= 0;

        app_en <= 1'b0;
        app_wdf_wren <= 1'b0;
        app_wdf_end <= 1'b0;
        app_cmd <= 3'b1;
        app_addr <= 27'h0;
        app_wdf_data <= 128'h0;

        write_count <= 0;
        state <= idle;
    end
    else if(write_stb)
    begin
        if(state==idle)
        begin
            if(app_rdy & app_wdf_rdy)
            begin
                write_ack <= 0;

                app_en <= 1'b1;
                app_wdf_wren <= 1'b1;
```

```
                    app_wdf_end <= 1'b1;
                    app_cmd <= 3'b0;
                    app_addr <= write_addr;
                    app_wdf_data <= write_data;

                    write_count <= write_count + 1;
                    state <= write;
                end
                else
                begin
                    state <= idle;
                end
            end
            else if(state==write)
            begin
                write_ack <= 1;
                if(write_count == 3)
                begin
                    read_enable <= 1;
                end
                app_en <= 1'b0;
                app_wdf_wren <= 1'b0;
                app_wdf_end <= 0;
                app_cmd <= 3'b1;

                state <= idle;
            end
            else
            begin
                state <= idle;
            end
        end
        else
        begin
            write_ack <= 0;
            app_en <= 0;
            app_wdf_wren <= 0;
            app_wdf_end <= 0;

            state <= idle;
        end
    end
endmodule
```

5.4DDR 读控制模块

```
`timescale 1ns / 1ps
module ddr2_read_control(
    input clk,
    input reset,
    input enable,
    //ddr2 signals
    input app_rdy,
    output reg app_en,
    output reg [2:0] app_cmd
    );

    parameter idle = 2'b01;
    parameter read = 2'b10;

    reg [1:0] state;

    always @(posedge clk)
    begin
        if(reset)
        begin
            app_en <= 0;
            app_cmd <= 0;
            state <= idle;
        end
        else if(enable)
        begin
            if(state==idle)
            begin
                app_en <= 1;
                app_cmd <= 3'b001;
                state <= read;
            end
            else if(state==read)
            begin
                if(app_rdy)
                begin
                    app_en <= 0;
                    state <= idle;
                end
            end
            else
```

```
            begin
                state <= idle;
            end
        end
        else
        begin
            app_en <= 0;
            app_cmd <= 0;
            state <= idle;
        end
    end

endmodule
```

6、SD 模块

作用：保存数据，并将数据送入 DDR
输入：SD RAM 地址、管脚相关输入信号
输出：SD RAM 的数据、SD 初始化、读、写是否完成的信号、管脚相关输出信号

6.1SD 顶层模块

```
`timescale 1ns / 1ps
module topsd(
    input clk,
    input reset,
    input sd_miso,
    output sd_cs,
    output sd_mosi,
    output sd_clk,
//    output led,
//    output [7:0]o_seg,
//    output [7:0]o_sel,
    output init_done,
    output write_finish,
    output read_finish,

    input [8:0]raddr,
    output [7:0]rdata
    );
    assign write_finish = 1'b1;
```

```
//wire init_done;
wire write_start;
wire [31:0]write_addr;
wire [15:0]write_data;
wire write_busy;
wire write_request;
wire read_start;
wire read_enable;
wire [31:0]read_addr;
wire [15:0]read_data;
wire read_error;

wire [8:0]ram_addr;

//产生SD卡测试数据
data_gen u_data_gen(
    clk,
    reset,
    init_done,

    write_busy,
    write_request,
    write_start,
    write_addr,
    write_data,

    read_enable,
    read_data,
    read_start,
    read_addr,
    read_error,

    ram_addr
    );

//SD卡顶层控制模块
sd_ctrl_top u_sd_ctrl_top(
    clk,
    reset,
    //SD卡接口
    sd_miso,
    sd_clk,
    sd_cs,
    sd_mosi,
```

```
    //写 SD 卡接口
    write_start,
    write_addr,
    write_data,
    write_busy,
    write_request,
    //读 SD 卡接口
    read_start,
    read_addr,
    read_enable,
    read_data,
    //初始化完成
    init_done,

    read_finish
    );

////led 警示
//led_alarm u_led_alarm(
//    clk,
//    reset,
//    read_error,
//    led
//    );

//wire [31:0]rdata;

////7 段数码管
//seg7x16 seg(
//    clk,
//    reset,
//    rdata,
//    o_seg,
//    o_sel
//);

//ram
sd_ram ram(
    clk,
    read_enable,
    ram_addr,
    read_data,

    raddr,
```

```
    rdata
);

endmodule
```

6.2SD 数据和地址产生模块

```
`timescale 1ns / 1ps
module data_gen(
    input clk,
    input reset,
    input init_done,//SD 卡初始化完成信号
    //写 SD 卡接口
    input write_busy,//写数据忙信号
    input write_request,//写数据请求信号
    output reg write_start,//开始写 SD 卡数据信号
    output reg [31:0]write_addr,//写数据扇区地址
    output [15:0]write_data,//写数据
    //读 SD 卡接口
    input read_enable,//读数据有效信号
    input [15:0]read_data,//读数据
    output reg read_start,//开始读 SD 卡数据信号
    output reg [31:0]read_addr,//读数据扇区地址
    output read_error,//SD 卡读写错误的标志

    output reg [8:0]ram_addr
    );

reg init_done_beat1;//init_done 信号延时打拍
reg init_done_beat2;
reg write_busy_beat1;//write_busy 信号延时打拍
reg write_busy_beat2;
reg [15:0]reg_write_data;
reg [15:0]comp_read_data;//用于对读出数据作比较的正确数据
reg [8:0]right_read_cnt;//读出正确数据的个数

wire pos_init_done;//init_done 信号的上升沿,用于启动写入信号
wire neg_write_busy;//write_busy 信号的下降沿,用于判断数据写入完成

assign pos_init_done = (~init_done_beat2) & init_done_beat1;

assign neg_write_busy = write_busy_beat2 & (~write_busy_beat1);
```

```
assign write_data = reg_write_data;

assign read_error = (right_read_cnt == (9'd256)) ? 1'b0 : 1'b1;//读128次正确的数据,说明读写测试成功,read_error = 0

reg [15:0] ins[0:1023];//指令寄存器模块
reg [15:0]ins_cnt;
initial
begin
    $readmemh("F:/Download/2.hex.txt", ins);
end

//信号延时打拍
always @(posedge clk or negedge reset)
begin
    if(!reset)
    begin
        init_done_beat1 <= 1'b0;
        init_done_beat2 <= 1'b0;
        write_busy_beat1 <= 1'b0;
        write_busy_beat2 <= 1'b0;
    end
    else
    begin
        init_done_beat1 <= init_done;
        init_done_beat2 <= init_done_beat1;
        write_busy_beat1 <= write_busy;
        write_busy_beat2 <= write_busy_beat1;
    end
end

//SD卡写入信号控制
always @(posedge clk or negedge reset)
begin
    if(!reset)
    begin
        write_start <= 1'b0;
        write_addr <= 32'd0;
    end
    else
    begin
        if(pos_init_done)
        begin
            write_start <= 1'b1;
```

```
            write_addr <= 32'd20;//指定扇区地址 2000
        end
        else
            write_start <= 1'b0;
    end
end

//SD卡写数据
always @(posedge clk or negedge reset)
begin
    if(!reset)
    begin
        reg_write_data <= 16'b0;
        ins_cnt <= 16'b0;
    end
    else
    begin
        if(write_request)
        begin
            ins_cnt <= ins_cnt + 16'b1;
            if (ins_cnt < 106)
            begin
                reg_write_data <= ins[ins_cnt];
            end
            else
            begin
                reg_write_data <= 16'b0;
            end
        end
    end
end

//SD卡读出信号控制
always @(posedge clk or negedge reset)
begin
    if(!reset)
    begin
        read_start <= 1'b0;
        read_addr <= 32'd0;
    end
    else
    begin
        if(neg_write_busy)
        begin
```

```
            read_start <= 1'b1;
            read_addr <= 32'd20;
        end
        else
            read_start <= 1'b0;
    end
end

//读数据错误时给出标志
always @(posedge clk or negedge reset)
begin
    if(!reset)
    begin
        comp_read_data <= 16'd0;
        right_read_cnt <= 9'd0;

        ram_addr <= 9'b0;
    end
    else
    begin
        if(read_enable)
        begin
            comp_read_data <= comp_read_data + 16'b1;
            if(ram_addr < 9'd212)
                ram_addr <= ram_addr + 9'd2;
            else
                ram_addr <= ram_addr;

            if(comp_read_data < 106)
            begin
                if(read_data == ins[comp_read_data])
                    right_read_cnt <= right_read_cnt + 9'd1;
            end
            else
            begin
                if(read_data == 16'b0)
                    right_read_cnt <= right_read_cnt + 9'd1;
            end
        end
    end
end

endmodule
```

6.3SD RAM 模块

```
module sd_ram(
    input clk,
    input we,
    input [8:0]waddr,
    input [15:0]wdata,

    input [8:0]raddr,
    output [7:0]rdata
    );

    reg[7:0]reg_ram[0:511];

    assign rdata = reg_ram[raddr];

    always @(posedge clk)
    begin
        if(we)
        begin
            reg_ram[waddr] <= wdata[15:8];
            reg_ram[waddr+1] <= wdata[7:0];
        end
    end

endmodule
```

6.4SD 控制顶层模块

```
`timescale 1ns / 1ps
module sd_ctrl_top(
    input clk,
    input reset,
    //SD 卡接口
    input sd_miso,
    output sd_clk,
    output sd_cs,
    output sd_mosi,
    //用户写 SD 卡接口
    input write_start,
    input [31:0]write_addr,
    input [15:0]write_data,
    output write_busy,
```

```
    output write_request,
    //用户读 SD 卡接口
    input read_start,
    input [31:0]read_addr,
    output read_enable,//读数据有效信号
    output [15:0]read_data,
    output init_done,

    output read_finish
    );

wire read_busy;
wire init_sd_clk;
wire init_sd_cs;
wire init_sd_mosi;
wire write_sd_cs;
wire write_sd_mosi;
wire read_sd_cs;
wire read_sd_mosi;

assign sd_clk = (init_done==1'b0) ? init_sd_clk : ~clk; //时钟信号,与 clk_ref 相
位相差 180 度

//SD 卡接口信号选择
assign sd_cs =  (init_done == 1'b0) ? init_sd_cs :
                (write_busy) ? write_sd_cs :
                (read_busy) ? read_sd_cs : 1'b1;

assign sd_mosi= (init_done == 1'b0) ? init_sd_mosi :
                (write_busy) ? write_sd_mosi :
                (read_busy) ? read_sd_mosi : 1'b1;

//SD 卡初始化
sd_init u_sd_init(
    clk,
    reset,
    sd_miso,
    init_sd_clk,
    init_sd_cs,
    init_sd_mosi,
    init_done
    );

//SD 卡写数据
```

```
sd_write u_sd_write(
    clk,
    reset,
    sd_miso,
    write_sd_cs,
    write_sd_mosi,
    //SD卡初始化完成之后响应写操作
    write_start & init_done,
    write_addr,
    write_data,
    write_busy,
    write_request
    );

//SD卡读数据
sd_read u_sd_read(
    clk,
    reset,
    sd_miso,
    read_sd_cs,
    read_sd_mosi,
    //SD卡初始化完成之后响应读操作
    read_start & init_done,
    read_addr,
    read_busy,
    read_enable,
    read_data,

    read_finish
    );

endmodule
```

6.5SD初始化模块

```
`timescale 1ns / 1ps
module sd_init(
    input clk,
    input reset,
    input sd_miso,
    output sd_clk,
    output reg sd_cs,
    output reg sd_mosi,
```

```
    output reg init_done
    );

parameter CMD0  = {8'h40,8'h00,8'h00,8'h00,8'h00,8'h95};//SD 卡软件复位命令

parameter CMD8  = {8'h48,8'h00,8'h00,8'h01,8'haa,8'h87};//发送主设备的电压范围

parameter CMD55 = {8'h77,8'h00,8'h00,8'h00,8'h00,8'hff};//告诉 SD 卡接下来的命令是应用相关命令

parameter  ACMD41  =  {8'h69,8'h40,8'h00,8'h00,8'h00,8'hff};//发送操作寄存器(OCR)内容

parameter div_num = 400;//时钟分频系数,初始化 SD 卡时降低 SD 卡的时钟频率,50M/250K = 200

parameter  wait_num  =  200;//上电至少等待 74 个同步时钟周期,在等待上电稳定期间,sd_cs = 1,sd_mosi = 1

parameter over_num = 25000;//发送软件复位命令时等待 SD 卡返回的最大时间,T = 100ms; 100_000us/4us = 25000

parameter to_wait = 7'b000_0001;         //默认状态,上电等待 SD 卡稳定
parameter send_cmd0 = 7'b000_0010;   //发送软件复位命令
parameter wait_cmd0 = 7'b000_0100;   //等待 SD 卡响应
parameter send_cmd8 = 7'b000_1000;   //发送主设备的电压范围，检测 SD 卡是否满足
parameter send_cmd55 = 7'b001_0000;  //告诉 SD 卡接下来的命令是应用相关命令
parameter send_acmd41 = 7'b010_0000; //发送操作寄存器(OCR)内容
parameter _init_done = 7'b100_0000;   //SD 卡初始化完成

reg [7:0]now_state;
reg [7:0]next_state;
reg [7:0]div_cnt;           //分频计数器
reg div_clk;                //分频后的时钟
reg [12:0]wait_cnt;      //上电等待稳定计数器

reg res_enable;                 //接收 SD 卡返回数据有效信号
reg [47:0]res_data;         //接收 SD 卡返回数据
reg res_flag;               //开始接收返回数据的标志
reg [5:0]res_bit_cnt;       //接收位数据计数器

reg [5:0]cmd_bit_cnt;       //发送指令位计数器
reg [15:0]over_time_cnt;    //超时计数器
reg  over_time_enable;          //超时使能信号
```

```
assign sd_clk = ~div_clk;  //SD_CLK,相位和DIV_CLK相差180度的时钟

//时钟分频,div_clk = 250KHz
always @(posedge clk or negedge reset)
begin
    if(!reset)
    begin
        div_clk <= 1'b0;
        div_cnt <= 8'd0;
    end
    else
    begin
        if(div_cnt == div_num/2-1'b1)
        begin
            div_clk <= ~div_clk;
            div_cnt <= 8'd0;
        end
        else
            div_cnt <= div_cnt + 1'b1;
    end
end

//上电等待稳定计数器
always @(posedge div_clk or negedge reset)
begin
    if(!reset)
        wait_cnt <= 13'd0;
    else if(now_state == to_wait)
    begin
        if(wait_cnt < wait_num)
            wait_cnt <= wait_cnt + 1'b1;
    end
    else
        wait_cnt <= 13'd0;
end

//接收sd卡返回的响应数据,在sd_clk的上升沿锁存数据
always @(posedge sd_clk or negedge reset)
begin
    if(!reset)
    begin
        res_enable <= 1'b0;
        res_data <= 48'd0;
```

```
        res_flag <= 1'b0;
        res_bit_cnt <= 6'd0;
    end
    else
    begin
        if(sd_miso == 1'b0 && res_flag == 1'b0) //sd_miso = 0 开始接收响应数据
        begin
            res_flag <= 1'b1;
            res_data <= {res_data[46:0],sd_miso};
            res_bit_cnt <= res_bit_cnt + 6'd1;
            res_enable <= 1'b0;
        end
        else if(res_flag)
        begin
            res_data <= {res_data[46:0],sd_miso}; //R1 返回 1 个字节,R3 R7 返回 5
个字节,在这里统一按照 6 个字节来接收,多出的 1 个字节为 NOP(8 个时钟周期的延时)
            res_bit_cnt <= res_bit_cnt + 6'd1;
            if(res_bit_cnt == 6'd47)
            begin
                res_flag <= 1'b0;
                res_bit_cnt <= 6'd0;
                res_enable <= 1'b1;
            end
        end
        else
            res_enable <= 1'b0;
    end
end

//状态转移
always @(posedge div_clk or negedge reset)
begin
    if(!reset)
        now_state <= to_wait;
    else
        now_state <= next_state;
end

//
always @(*)
begin
    next_state = to_wait;
    case(now_state)
        to_wait :
```

```
        begin
            if(wait_cnt == wait_num)
                next_state = send_cmd0;
            else
                next_state = to_wait;
        end
        send_cmd0 :
        begin
            if(cmd_bit_cnt == 6'd47)
                next_state = wait_cmd0;
            else
                next_state = send_cmd0;
        end
        wait_cmd0 :
        begin
            if(res_enable) //SD卡返回响应信号
            begin
                if(res_data[47:40] == 8'h01) //SD卡返回复位成功
                    next_state = send_cmd8;
                else
                    next_state = to_wait;
            end
            else if(over_time_enable) //SD卡响应超时
                next_state = to_wait;
            else
                next_state = wait_cmd0;
        end
        send_cmd8 :
        begin
            if(res_enable) //SD卡返回响应信号
            begin
                if(res_data[19:16] == 4'b0001) //返回SD卡的操作电压,[19:16] =
4'b0001(2.7V~3.6V)
                    next_state = send_cmd55;
                else
                    next_state = to_wait;
            end
            else
                next_state = send_cmd8;
        end

        send_cmd55 :
        begin
            if(res_enable) //SD卡返回响应信号
```

```
            begin
                if(res_data[47:40] == 8'h01) //SD卡返回空闲状态
                    next_state = send_acmd41;
                else
                    next_state = send_cmd55;
            end
            else
                next_state = send_cmd55;
        end
        send_acmd41 :
        begin
            if(res_enable) //SD卡返回响应信号
            begin
                if(res_data[47:40] == 8'h00) //初始化完成信号
                    next_state = _init_done;
                else
                    next_state = send_cmd55; //初始化未完成,重新发起
            end
            else
                next_state = send_acmd41;
        end
        _init_done : next_state = _init_done; //初始化完成
        default : next_state = to_wait;
    endcase
end

//SD卡在 sd_clk 的下降沿输出数据,为了统一在 alway 块中使用上升沿触发,此处使用和
sd_clk 相位相差180度的时钟
always @(posedge div_clk or negedge reset)
begin
    if(!reset)
    begin
        sd_cs <= 1'b1;
        sd_mosi <= 1'b1;
        init_done <= 1'b0;
        cmd_bit_cnt <= 6'd0;
        over_time_cnt <= 16'd0;
        over_time_enable <= 1'b0;
    end
    else
    begin
        over_time_enable <= 1'b0;
        case(now_state)
            to_wait :
```

```
begin
    sd_cs <= 1'b1;
    sd_mosi <= 1'b1;
end
send_cmd0 : //发送 CMD0 软件复位命令
begin
    cmd_bit_cnt <= cmd_bit_cnt + 6'd1;
    sd_cs <= 1'b0;
    sd_mosi <= CMD0[6'd47 - cmd_bit_cnt];//先发送 CMD0 命令高位
    if(cmd_bit_cnt == 6'd47)
        cmd_bit_cnt <= 6'd0;
end
wait_cmd0 : //在接收 CMD0 响应返回期间,片选 CS 拉低,进入 SPI 模式
begin
    sd_mosi <= 1'b1; //SD 卡返回响应信号
    if(res_enable) //接收完成之后再拉高,进入 SPI 模式
        sd_cs <= 1'b1;
    over_time_cnt <= over_time_cnt + 1'b1; //超时计数器开始计数
    if(over_time_cnt == over_num - 1'b1) //SD 卡响应超时,重新发送软
件复位命令
        over_time_enable <= 1'b1;
    if(over_time_enable)
        over_time_cnt <= 16'd0;
end
send_cmd8 : //发送 CMD8
begin
    if(cmd_bit_cnt<=6'd47)
    begin
        cmd_bit_cnt <= cmd_bit_cnt + 6'd1;
        sd_cs <= 1'b0;
        sd_mosi <= CMD8[6'd47 - cmd_bit_cnt]; //先发送 CMD8 命令高位
    end
    else
    begin
        sd_mosi <= 1'b1;
        if(res_enable) //SD 卡返回响应信号
        begin
            sd_cs <= 1'b1;
            cmd_bit_cnt <= 6'd0;
        end
    end
end
send_cmd55 : //发送 CMD55
begin
```

```
    if(cmd_bit_cnt<=6'd47)
    begin
        cmd_bit_cnt <= cmd_bit_cnt + 6'd1;
        sd_cs <= 1'b0;
        sd_mosi <= CMD55[6'd47 - cmd_bit_cnt];
    end
    else
    begin
        sd_mosi <= 1'b1;
        if(res_enable) //SD卡返回响应信号
        begin
            sd_cs <= 1'b1;
            cmd_bit_cnt <= 6'd0;
        end
    end
end
send_acmd41 : //发送ACMD41
begin
    if(cmd_bit_cnt <= 6'd47)
    begin
        cmd_bit_cnt <= cmd_bit_cnt + 6'd1;
        sd_cs <= 1'b0;
        sd_mosi <= ACMD41[6'd47 - cmd_bit_cnt];
    end
    else
    begin
        sd_mosi <= 1'b1;
        if(res_enable) //SD卡返回响应信号
        begin
            sd_cs <= 1'b1;
            cmd_bit_cnt <= 6'd0;
        end
    end
end
_init_done : //初始化完成
begin
    init_done <= 1'b1;
    sd_cs <= 1'b1;
    sd_mosi <= 1'b1;
end
default :
begin
    sd_cs <= 1'b1;
    sd_mosi <= 1'b1;
```

```
            end
        endcase
    end
end

endmodule
```

6.6SD 写模块

```
`timescale 1ns / 1ps
module sd_write(
    input clk,
    input reset,
    //SD 卡接口
    input sd_miso,
    output reg sd_cs,
    output reg sd_mosi,
    //写接口
    input write_start,
    input [31:0]write_addr,
    input [15:0]write_data,
    output reg write_busy,
    output reg write_request
    );

parameter HEAD_BYTE = 8'hfe;//数据头

reg write_enable_beat1;//write_start 信号延时打拍
reg write_enable_beat2;

reg res_enable;              //接收 SD 卡返回数据有效信号
reg [7:0]res_data;       //接收 SD 卡返回数据
reg res_flag;            //开始接收返回数据的标志
reg [5:0]res_bit_cnt;    //接收位数据计数器

reg [3:0]write_state_cnt;    //写控制计数器
reg [47:0]cmd_write;         //写命令
reg [5:0]cmd_bit_cnt;    //写命令位计数器
reg [3:0]data_bit_cnt;         //写数据位计数器,16
reg [8:0]data_cnt;       //写入数据数量,256
reg [15:0]reg_write_data;     //寄存写入的数据，防止发生改变
reg detect_done_flag;    //检测写空闲信号的标志
reg [7:0]detect_data;    //检测到的数据
```

```
wire pos_write_enable;//开始写SD卡数据信号的上升沿

assign pos_write_enable = (~write_enable_beat2) & write_enable_beat1;

//信号延时打拍
always @(posedge clk or negedge reset)
begin
    if(!reset)
    begin
        write_enable_beat1 <= 1'b0;
        write_enable_beat2 <= 1'b0;
    end
    else
    begin
        write_enable_beat1 <= write_start;
        write_enable_beat2 <= write_enable_beat1;
    end
end

//接收sd卡返回的响应数据,在clk的下降沿锁存数据
always @(negedge clk or negedge reset)
begin
    if(!reset)
    begin
        res_enable <= 1'b0;
        res_data <= 8'd0;
        res_flag <= 1'b0;
        res_bit_cnt <= 6'd0;
    end
    else
    begin
        if(sd_miso == 1'b0 && res_flag == 1'b0) //sd_miso = 0 开始接收响应数据
        begin
            res_flag <= 1'b1;
            res_data <= {res_data[6:0],sd_miso};
            res_bit_cnt <= res_bit_cnt + 6'd1;
            res_enable <= 1'b0;
        end
        else if(res_flag)
        begin
            res_data <= {res_data[6:0],sd_miso};
            res_bit_cnt <= res_bit_cnt + 6'd1;
            if(res_bit_cnt == 6'd7)
```

```
            begin
                res_flag <= 1'b0;
                res_bit_cnt <= 6'd0;
                res_enable <= 1'b1;
            end
        end
        else
            res_enable <= 1'b0;
    end
end

//写完数据后检测 SD 卡是否空闲
always @(posedge clk or negedge reset)
begin
    if(!reset)
        detect_data <= 8'd0;
    else if(detect_done_flag)
        detect_data <= {detect_data[6:0],sd_miso};
    else
        detect_data <= 8'd0;
end

//SD 卡写入数据
always @(posedge clk or negedge reset)
begin
    if(!reset)
    begin
        sd_cs <= 1'b1;
        sd_mosi <= 1'b1;
        write_state_cnt <= 4'd0;
        write_busy <= 1'b0;
        cmd_write <= 48'd0;
        cmd_bit_cnt <= 6'd0;
        data_bit_cnt <= 4'd0;
        reg_write_data <= 16'd0;
        data_cnt <= 9'd0;
        write_request <= 1'b0;
        detect_done_flag <= 1'b0;
    end
    else
    begin
        write_request <= 1'b0;
        case(write_state_cnt)
            4'd0 : //写空闲
```

```
            begin
                write_busy <= 1'b0;
                sd_cs <= 1'b1;
                sd_mosi <= 1'b1;
                if(pos_write_enable)
                begin
                    cmd_write  <=  {8'h58,write_addr,8'hff}; //写入单个命令块
CMD24
                    write_state_cnt <= write_state_cnt + 4'd1; //控制计数器加 1
                    write_busy <= 1'b1;//开始执行写入数据,拉高写忙信号
                end
            end
            4'd1 :
            begin
                if(cmd_bit_cnt <= 6'd47) //开始按位发送写命令
                begin
                    cmd_bit_cnt <= cmd_bit_cnt + 6'd1;
                    sd_cs <= 1'b0;
                    sd_mosi <= cmd_write[6'd47 - cmd_bit_cnt]; //先发送高字节
                end
                else
                begin
                    sd_mosi <= 1'b1;
                    if(res_enable) //SD 卡响应
                    begin
                        write_state_cnt <= write_state_cnt + 4'd1; //控制计数器
加 1
                        cmd_bit_cnt <= 6'd0;
                        data_bit_cnt <= 4'd1;
                    end
                end
            end
            4'd2 :
            begin
                data_bit_cnt <= data_bit_cnt + 4'd1;
                //data_bit_cnt = 0~7 等待 8 个时钟周期
                //data_bit_cnt = 8~15,写入命令头 8'hfe
                if(data_bit_cnt>=4'd8 && data_bit_cnt <= 4'd15)
                begin
                    sd_mosi <= HEAD_BYTE[4'd15-data_bit_cnt]; //先发送高字节
                    if(data_bit_cnt == 4'd14)
                        write_request <= 1'b1; //提前拉高写数据请求信号
                    else if(data_bit_cnt == 4'd15)
                        write_state_cnt <= write_state_cnt + 4'd1; //控制计数器
```

加 1

```
                end
            end
            4'd3 : //写入数据
            begin
                data_bit_cnt <= data_bit_cnt + 4'd1; //bit_cnt 归零
                if(data_bit_cnt == 4'd0)
                begin
                    sd_mosi <= write_data[4'd15-data_bit_cnt]; //先发送数据高位
                    reg_write_data <= write_data; //寄存数据
                end
                else
                    sd_mosi <= reg_write_data[4'd15-data_bit_cnt]; //先发送数据
高位
                if((data_bit_cnt == 4'd14) && (data_cnt <= 9'd255))
                    write_request <= 1'b1;
                if(data_bit_cnt == 4'd15)
                begin
                    data_cnt <= data_cnt + 9'd1;
                    if(data_cnt == 9'd255) //写入单个 BLOCK 共 512 个字节 = 256 *
16bit
                    begin
                        data_cnt <= 9'd0;
                        write_state_cnt <= write_state_cnt + 4'd1;
                    end
                end
            end
            4'd4 : //写入两个字节的 8'hff 进行 CRC 校验
            begin
                data_bit_cnt <= data_bit_cnt + 4'd1;
                sd_mosi <= 1'b1;
                if(data_bit_cnt == 4'd15)
                    write_state_cnt <= write_state_cnt + 4'd1;
            end
            4'd5 :
            begin
                if(res_enable) //SD 卡响应
                    write_state_cnt <= write_state_cnt + 4'd1;
            end
            4'd6 : //等待写完成
            begin
                detect_done_flag <= 1'b1;
                if(detect_data == 8'hff) //detect_data = 8'hff 时,SD 卡写入完成,
进入空闲状态
```

```
                begin
                    write_state_cnt <= write_state_cnt + 4'd1;
                    detect_done_flag <= 1'b0;
                end
            end
            default :
            begin
                sd_cs <= 1'b1; //进入空闲状态后,拉高片选信号,等待 8 个时钟周期
                write_state_cnt <= write_state_cnt + 4'd1;
            end
        endcase
    end
end

endmodule
```

6.7SD 读模块

```
`timescale 1ns / 1ps
module sd_read(
    input clk,
    input reset,

    input sd_miso,
    output reg sd_cs,
    output reg sd_mosi,

    input read_start,
    input [31:0]read_addr,
    output reg read_busy,
    output reg read_enable,
    output reg [15:0]read_data,

    output reg read_finish
    );

reg read_beat1;
reg read_beat2;

reg res_enable;
reg [7:0]res_data;
reg res_flag;
reg [5:0]res_bit_cnt;
```

```
reg get_en_t;
reg [15:0]get_data_t;
reg get_flag;
reg [3:0]get_bit_cnt;
reg [8:0]get_data_cnt;
reg get_finish_en;

reg [3:0]rd_ctrl_cnt;
reg [47:0]read_cmd;
reg [5:0]cmd_bit_cnt;
reg rd_data_flag;

wire pos_rd_en;

assign  pos_rd_en = (~read_beat2) & read_beat1;

//信号延时打拍
always @(posedge clk or negedge reset)
begin
    if(!reset)
    begin
        read_beat1 <= 1'b0;
        read_beat2 <= 1'b0;
    end
    else
    begin
        read_beat1 <= read_start;
        read_beat2 <= read_beat1;
    end
end

//接收 sd 卡返回的响应数据,在 sd_clk 的上升沿锁存数据
always @(negedge clk or negedge reset)
begin
    if(!reset)
    begin
        res_enable <= 1'b0;
        res_data <= 8'd0;
        res_flag <= 1'b0;
        res_bit_cnt <= 6'd0;
    end
    else
    begin
```

```
        //sd_miso = 0 开始接收响应数据
        if(sd_miso == 1'b0 && res_flag == 1'b0)
        begin
            res_flag <= 1'b1;
            res_data <= {res_data[6:0],sd_miso};
            res_bit_cnt <= res_bit_cnt + 6'd1;
            res_enable <= 1'b0;
        end
        else if(res_flag)
        begin
            res_data <= {res_data[6:0],sd_miso};
            res_bit_cnt <= res_bit_cnt + 6'd1;
            if(res_bit_cnt == 6'd7)
            begin
                res_flag <= 1'b0;
                res_bit_cnt <= 6'd0;
                res_enable <= 1'b1;
            end
        end
        else
            res_enable <= 1'b0;
    end
end

//接收 SD 卡有效数据在 sd_clk 的上升沿锁存数据
always @(negedge clk or negedge reset)
begin
    if(!reset)
    begin
        get_en_t <= 1'b0;
        get_data_t <= 16'd0;
        get_flag <= 1'b0;
        get_bit_cnt <= 4'd0;
        get_data_cnt <= 9'd0;
        get_finish_en <= 1'b0;
    end
    else
    begin
        get_en_t <= 1'b0;
        get_finish_en <= 1'b0;
        if(rd_data_flag && sd_miso == 1'b0 && get_flag == 1'b0)//数据头 0xfe
8'b1111_1110，所以检测 0 为起始位
            get_flag <= 1'b1;
        else if(get_flag)
```

```
        begin
            get_bit_cnt <= get_bit_cnt + 4'd1;
            get_data_t <= {get_data_t[14:0],sd_miso};
            if(get_bit_cnt == 4'd15)
            begin
                get_data_cnt <= get_data_cnt + 9'd1;

                if(get_data_cnt <= 9'd255) //接收单个BLOCK共512个字节 = 256 *
16bit
                    get_en_t <= 1'b1;
                else if(get_data_cnt == 9'd257)
                begin   //接收两个字节的CRC校验值
                    get_flag <= 1'b0;
                    get_finish_en <= 1'b1;//数据接收完成
                    get_data_cnt <= 9'd0;
                    get_bit_cnt <= 4'd0;
                end
            end
        end
        else
            get_data_t <= 16'd0;
    end
end

//寄存输出数据有效信号和数据
always @(posedge clk or negedge reset)
begin
    if(!reset)
    begin
        read_enable <= 1'b0;
        read_data <= 16'd0;
    end
    else
    begin
        if(get_en_t)
        begin
            read_enable <= 1'b1;
            read_data <= get_data_t;
        end
        else
            read_enable <= 1'b0;
    end
end
```

```
//读命令
always @(posedge clk or negedge reset)
begin
    if(!reset)
    begin
        sd_cs <= 1'b1;
        sd_mosi <= 1'b1;
        rd_ctrl_cnt <= 4'd0;
        read_cmd <= 48'd0;
        cmd_bit_cnt <= 6'd0;
        read_busy <= 1'b0;
        rd_data_flag <= 1'b0;

        read_finish <= 1'b0;
    end
    else
    begin
        case(rd_ctrl_cnt)
            4'd0 :
            begin
                read_busy <= 1'b0;
                sd_cs <= 1'b1;
                sd_mosi <= 1'b1;
                if(pos_rd_en)
                begin
                    read_cmd <= {8'h51,read_addr,8'hff};
                    rd_ctrl_cnt <= rd_ctrl_cnt + 4'd1;
                    read_busy <= 1'b1;
                end
            end
            4'd1 :
            begin
                if(cmd_bit_cnt <= 6'd47)
                begin
                    cmd_bit_cnt <= cmd_bit_cnt + 6'd1;
                    sd_cs <= 1'b0;
                    sd_mosi <= read_cmd[6'd47 - cmd_bit_cnt];
                end
                else
                begin
                    sd_mosi <= 1'b1;
                    if(res_enable)
                    begin
                        rd_ctrl_cnt <= rd_ctrl_cnt + 4'd1;
```

```
                    cmd_bit_cnt <= 6'd0;
                end
            end
        end
        4'd2 :
        begin
            rd_data_flag <= 1'b1;
            if(get_finish_en)
            begin
                rd_ctrl_cnt <= rd_ctrl_cnt + 4'd1;
                rd_data_flag <= 1'b0;
                sd_cs <= 1'b1;

                read_finish <= 1'b1;
            end
        end
        default :
        begin
            sd_cs <= 1'b1;
            rd_ctrl_cnt <= rd_ctrl_cnt + 4'd1;
        end
    endcase
end
end

endmodule
```

六、实验体会

对于本次实验，首先的一点感受就是对于 Verilog 语言的理解更加深刻了。由于三级存储实验中涉及到很多控制信号、寄存器、状态机等的操作，因此，直接采用高级语言程序设计的思想去用 Verilog 语言进行底层逻辑电路的设计是完全行不通的。

在三级存储实验过程中，我通过查询网上资料，尤其是 Xilinx 官网提供的 SD 卡程序实例，了解了 SD 卡的 SPI 协议，然后从零实现了对 SD 工作流的设计，我发现 SD 卡的驱动是比较顺利的，通过调试状态机、输出初始化、读、写几部分的相关信号，较为顺利地完成了 SD 模块的设计。同时，我借助实验指导书以及课程群里给的 DDR 读写实例，自己对 DDR 的读写进行了摸索，但是我发现，在读写过程中会出现数据时对时不对的问题，然后我发现在实验指导书中提到“重复读取 256 次”，于是我也按照实验指导书中的要求，对于 DDR 连续地读取了 256 次，最终成功实现了 DDR 的正确读取，在这里我猜测可能是因为 MIG 这个 IP 核内部电路较复杂，门级较多，在 100MHZ 的频率下不能稳定运行。当 9FD0B7EA 这个答案稳定的出现在七段数码管上时，我如释重负，感觉自己的努力终究没有白费。

在实验过程中，我尽量采用了高效的、门级少的设计方式进行编码。在每一个模块设计过程中，也使得我对理论课程所学习的内容有了更深的理解和体会，通过这次实验，我对于三级存储系统的认识更加深刻了。

本次实验也大大增加了对 CPU、存储系统以及整个计算机系统结构的兴趣。其实我本来就对于计算机硬件比较感兴趣，并一直有志于为国产硬件设计的发展贡献出自己的一份力。在这次实验中，通过一次一次的尝试、一次一次的设计状态机，使我感觉到硬件语言设计过程的乐趣，并对于 CPU 以及存储系统设计中可能存在的问题与解决方案有了一个基本的了解。这也正是三级存储设计过程中一个非常有趣的地方。虽然花费了不少的时间，但是有着巨大的收获，非常值得。